Final Report

Advanced Hard Real-Time Operating System, The Maruti Project

DASG-60-92-C-0055

To DOD - Army - Huntsville and Phillips Lab
Phillips Laboratory / PKVC
2251 Maxwell Street SE
Kirtland AFB NM 87117-5773

Ashok K. Agrawala and Satish K. Tripathi
Department of Computer Science
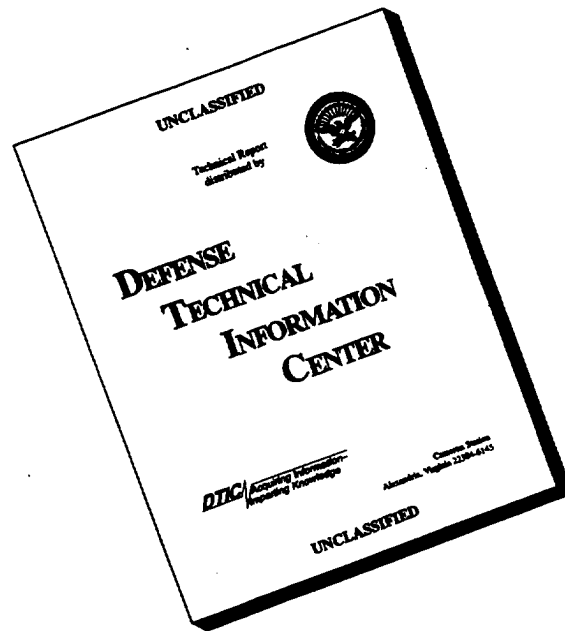University of Maryland
College Park, MD 20742
January 1997

DTIC QUALITY INSPECTED 3

19970130 010

DISCLAIMER NOTICE

UNCLASSIFIED

Technical Report
distributed by

**DEFENSE
TECHNICAL
INFORMATION
CENTER**

DTIC | Acquiring Information
Imparting Knowledge

Cameron Station
Alexandria, Virginia 22304-6145

UNCLASSIFIED

THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

TABLE OF CONTENTS

Executive Summary

Introduction:

This is the final report on the work done under contract DASG-60-92-C-0055 from Phillips Labs and ARPA to the Department of Computer Science at the University of Maryland. The work started 04/28/92. The goal of this project was to create an environment for development and deployment of critical applications with hard real-time constraints in a reactive environment. We have redesigned Maruti system to address these issues. In this report we highlight the achievements of this contract. A publications list and a copy of each of the publications is also attached.

Application Development Environment:

To support applications in a real-time system, conventional application development techniques and tools must be augmented with support for specification and extraction of resource requirements and timing constraints. The application development system provides a set of programming tools to support and facilitate the development of real-time applications with diverse requirements. The Maruti Programming Language (MPL) is used to develop induvidual program modules. The Maruti Configuration Language (MCL) is used to specify how individual program modules are to be connected together to form an application and the details of the hardware of which the application is to be executed.

In the current version, the base programming language used is ANSI C. MPL adds *modules, shared memory blocks, critical regions, typed message passing, periodic functions*, and *message-invoked functions* to the C language. To make analyzing the resource usage of programs feasible, certain C idioms are not allowed in MPL; in particular, recursive function calls are not allowed nor are unbounded loops containing externally visible events, such as message passing and critical region transition.

MPL Modules are brought together into as an executable application by a specification file written in the Maruti Configuration Language (MCL). The MCL specification determines the application's hard real-time constraints, the allocation of tasks, threads, and shared memory blocks, and all message-passing connections. MCL is an interpreted C-like language rather than a declarative language, allowing the instantiation of complicated subsystems using loops and subroutines in the specification.

Analysis and Resource Allocations:

The basic building block of the Maruti computation model is the elemental unit (EU). In general an elemental unit is an executable entity which is triggered by incoming data and signals, operates on the input data, and produces some output data and signals. The behavior of an EU is atomic with respect to its environment. Specifically:

- All resources needed by an elemental unit are assumed to be required for the entire length of its execution.
- The interaction of an EU with other entities of the system occurs either before it starts executing or after it finishes execution.

In order to define complex executions , the EUs may be composed together and properties specified on the composition. Elemental units are composed by connecting an output port of an EU with an input port of another EU. A valid connection requires that the input and output of port types are compatible, i.e., they carry the same message type. Such a connection marks a one-way flow of data or control, depending on the nature of the ports. A composition of EUs can be viewed as a directed acyclic graph, called an *elemental unit graph* (EUG), in which the nodes are the EUs, and the edges are the connections between EUs. An incompletely specified EUG in which all input and output ports are not connected is termed as a *partial* EUG (PEUG). A partial EUG may be viewed as a higher level EU. In a complete EUG, all input and output ports are connected and there are no cycles in the graph. The acyclic requirements come from the required time determinacy of execution. A program with unbounded cycles or recursions may not have a temporally determinate execution time. Bounded cycles in an EUG are converted into a acyclic graph by loop unrolling.

Program modules are independently compiled. In addition to the generation of the object code, compilation also results in the creation of partial EUGs for the modules, i.e., for the services and entries in the module, as well as the extraction of resource requirements such as stack sizes or threads, memory requirements, and the logical resource requirements.

Given an application specification in the Maruti Configuration Language and the component application modules, the integration tools are responsible for creating a complete application program and extracting out the resource and timing information for scheduling and resource allocation. The input of the integration process are the program modules, the partial EUGs corresponding to the modules, the application configuration specification, and the hardware specifications. The outputs of the integration process are: a specification for the loader for creating tasks, populating their address space, creating the threads and channels, and initializing the task; loadable executables of the program; and the complete application EUG along with the resource description for the resource allocation and the scheduling subsystem.

After the application program has been analyzed and its resource requirements and execution constraints identified, it can be allocated and scheduled for a runtime system.

We consider the static allocation and scheduling in which a task is the finest granularity object of allocation and an EU instance is the unit of scheduling. In order to make the execution of instances satisfy the specification and meet the timing constraints, we consider a scheduling frame whose length is the least common multiple of all tasks' periods. As long as one instance of each EU is scheduled in each period within the scheduling frame and these executions meet the timing constraints, a feasible schedule is obtained.

Maruti Runtime System:

The runtime system provides the conventional functionality of an operating system in a manner that supports the timely dispatching of jobs. There are two major components of the runtime system - the *Maruti core*, which is the operating system code that implements scheduling, message passing, process control, thread control, and low level hardware control, and the *runtime dispatcher*, which performs resource allocation and scheduling or dynamic arrivals.

The core of the Maruti hard real-time runtime system consists of three data structures:

- The *calendars* are created and loaded by the dispatcher. Kernel memory is reserved for each calendar at the time it is created. Several system calls serve to create, delete, modify, activate, and deactivate calendars.

- The *results table* holds timing and status results for the execution of each elemental unit; The maruti_calandar_results system call reports these results back up to the user level, usually the dispatcher. The dispatcher can then keep statistics or write a trace file.

- The *pending activation table* holds all outstanding calendar activation and deactivation requests. Since the requests can come from before the switch time, the kernel must track the requests and execute them at the correct time in the correct order.

The Maruti design includes the concept of scenarios, implemented at runtime as sets of alternative calendars that can be switched quickly to handle an emergency or a change in operating mode. These calendars are pre-scheduled and able to begin execution without having to invoke any user-level machinery. The dispatcher loads the initial scenarios specified by the application and activates one of them to begin normal execution.

Optimal Replication of Series-Parallel Graphs for Computation-Intensive Applications*

Sheng-Tzong Cheng
Ashok K. Agrawala
Institute for Advanced Computer Studies
Systems Design and Analysis Group
Department of Computer Science
University of Maryland, College Park, MD 20742
{stcheng,agrawala}@cs.umd.edu

1

Optimal Replication of SP Graphs for Computation-Intensive Applications

Prof. Ashok K. Agrawala

Department of Computer Science

University of Maryland at College Park

A.V. Williams Building

College Park, Maryland 20742

(Tel): (301) 405-2525

Abstract

We consider the replication problem of series-parallel (SP) task graphs where each task may run on more than one processor. The objective of the problem is to minimize the total cost of task execution and interprocessor communication. We call it, the *minimum cost replication problem for SP graphs* (MCRP-SP). In this paper, we adopt a new communication model where the purpose of replication is to reduce the total cost. The class of applications we consider is computation-intensive applications in which the execution cost of a task is greater than its communication cost. The complexity of MCRP-SP for such applications is proved to be NP-complete. We present a branch-and-bound method to find an optimal solution as well as an approximation approach for suboptimal solution. The numerical results show that such replication may lead to a lower cost than the optimal assignment problem (in which each task is assigned to only one processor) does. The proposed optimal solution has the complexity of $O(n^2 2^n M)$, while the approximation solution has $O(n^4 M^2)$, where n is the number of processors in the system and M is the number of tasks in the graph.

1 Introduction

Distributed computer systems have often resulted in improved reliability, flexibility, throughput, fault tolerance and resource sharing. In order to use the processors available in a distributed system, the tasks have to be allocated to the processors. The allocation problem is one of the basic problems of distributed computing whose solution has a far reaching impact on the usability and efficiency of a distributed system. Clearly, the tasks of an application have to be executed satisfying the precedence and other synchronization constraints among them. (Such constraints are often specified in the form of a task graph.)

In executing an application, defined by its task graph, we have the option of restricting ourselves to having only one copy of each task. The allocation problem, in this case, is referred to as *assignment problem*. If, on the other hand, a task may be replicated multiple times, the general problem is called the *replication problem*. In this paper, we consider the replication problem and present an algorithm to find the optimal replication of series-parallel graphs for computation-intensive applications.

For distributed processing applications, the objective of the allocation problem may be the minimum completion time, processor load balancing, or total cost of execution and communication, etc. For the assignment problem where the objective is to minimize the total cost of execution and interprocessor communication, Stone [11] and Towsley [12] presented $O(n^3 M)$ algorithms for tree-structure and series-parallel graphs, respectively, of M tasks and n processors. For general task graphs, the assignment problem has been proven [9] to be NP-complete. Many papers [8][9][10] presented branch-and-bound methods which yielded an optimal result. Other heuristic methods have been considered by Lo [7] and Price and Krishnaprasad [5]. All these works focused on the assignment problem.

Traditionally, the main purpose of replicating a task on multiple processors is to increase the degree of fault tolerance [2][6]. If some processors in the distributed system fail, the application may still survive using other copies. In such a communication model, a task has to communicate with multiple copies of other tasks. As a consequence, the total cost of execution and communication of the replication problem will be bigger than that of the assignment problem. In this paper, we adopt another communication model in which the replication of a task is not for the sake of fault tolerance but for decreasing of the total cost. In our model, each task may have more than one copy

and it may start its execution after receiving necessary data from one copy of each preceding task. Clearly, in a heterogeneous environment the cost of execution of a task depends on the processor on which it executes, and the communication costs depend on the topology, communication medium, protocols used, etc. When a task t is allowed to have only one copy in the system, the sum of the interprocessor communication costs between t and other tasks may be large. Sometimes it will be more beneficial if we replicate t onto multiple processors to reduce the inter-processor communication, and to fully utilize the available processors in the systems. Such replication may lead to a lower total cost than the optimal assignment problem does. An example illustrating this point is presented in Section 3.

In the assignment problem, polynomial-time algorithms exist for special cases, such as tree-structure [11] and series-parallel [12] task graphs. This paper represents one of the first few attempts at finding special cases for the replication problem. The class of applications we consider in this paper is computation-intensive applications in which the execution cost of a task is greater than its communication cost. Such applications can be found in an enormous number of fields, such as digital signal processing, weather forecasting, game searching, etc. We formally define a computation-intensive application in Section 2.2. In this paper, we prove that for the computation-intensive applications, the replication problem is NP-complete, and we present a branch-and-bound algorithm to solve it. The worst-case complexity of the solution is $O(n^2 2^n M)$. Note that the algorithm is able to solve the problem in the complexity of the linear function of M.

We also develop an approximation approach to solve the problem in polynomial time. Given a forker task s with K successors in the SP graph, the method tries to allocate s to processors based on iterative selection. The complexity of the iterative selection for a forker is $O(n^2 K^2)$, while the overall solution for an SP graph is $O(n^4 M^2)$.

In the remainder of this paper, the series-parallel graph model and the computation model are described in section 2. In section 3, the replication problem is formulated as the minimum cost 0-1 integer programming problem and the proof of NP completeness is given. A branch-and-bound algorithm and numerical results are given in section 4, while the approximation methods and results are given in section 5. The overall algorithm is presented and conclusion remark is drawn in section 6.

2 Definitions

2.1 Graph Model

A *series-parallel* (SP) graph, $G = (V, E)$, is a directed graph of type p, where $p \in \{T_{unit}, T_{chain}, T_{and}, T_{or}\}$ and G has a source node (of indegree 0) and a sink node (of outdegree 0). An SP graph can be constructed by applying the following rules recursively.

1. A graph $G = (V, E) = (\{v\}, \phi)$ is an SP graph of type T_{unit}. (Node v is the source and the sink of G.)

2. If $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are SP graphs then $G' = (V', E')$ is an SP graph of type T_{chain}, where $V' = V_1 \cup V_2$ and $E' = E_1 \cup E_2 \cup \{<\text{sink of } G_1, \text{source of } G_2 >\}$.

3. If each graph $G_i = (V_i, E_i)$ with source-sink pair (s_i, t_i), where s_i is of outdegree 1, is an SP graph, $\forall i = 1, 2, \ldots, n$, and new nodes $s' \notin V_i$ and $t' \notin V_i$, $\forall i$ are given then $G' = (V', E')$ is an SP graph of type T_{and} (or type T_{or}), where $V' = V_1 \cup V_2 \cup \ldots \cup V_n \cup \{s', t'\}$ and $E' = E_1 \cup E_2 \cup \ldots \cup E_n \cup \{< s', s_i > \mid \forall i = 1, 2, \ldots, n\} \cup \{< t_i, t' > \mid \forall i = 1, 2, \ldots, n\}$. The source of G', s', is called the *forker* of G'. The sink of G', t', is called the *joiner* of G'. G' is an SP graph of type T_{and} (or type T_{or}) if there exists a *parallel-and* (or *parallel-or*) relation among G_i's.

A convenient way of representing the structure of an SP graph is via a parsing tree [4]. The transformation of an SP graph to a parsing tree can be done in a recursive way. There are four kinds of internal nodes in a parsing tree: T_{unit}, T_{chain}, T_{and} and T_{or} nodes. A T_{unit} node has only one child, while a T_{chain} node has more than one child. Every internal node x, along with all its descendant nodes induces a subtree S_x which describes an SP subgraph G_x of G. Each leaf node in S_x corresponds to an SP graph of type T_{unit}. A T_{and} (or T_{or}) node y consists of its type T_{and} (or T_{or}) along with the forker and joiner nodes of G_y. We give an example of an SP graph G, and its parsing tree $T(G)$ in Figure 1.

2.2 Computational Model

An application program consists of M tasks labeled $m = 1, 2, \ldots, M$. Its behavior is represented by an SP graph with the tasks correspond to the nodes. Each task may be replicated onto more than one processor. A *task instance* $t_{i,p}$ is a replication of task i on processor p. A directed edge $< i, j >$ between nodes i and j exists if the execution of task j follows that of task i. Associated with each edge $< i, j >$ is the communication cost incurred by the application. We are concerned with types of applications where the cost of execution of a task is always greater than the communication overhead it needs. The model is stated as follows.

Given a distributed system S with n processors connected by a communication network, an application is *computation-intensive* if its associated SP graph $G = (V, E)$ on S satisfies the following conditions:

1. $\mu_{i,j}(p, q) \geq 0$,

2. $\sum_{q=1}^{n} \mu_{i,j}(p, q) \leq \min_p(e_{i,p})$, $\forall < i, j > \in E$, and $1 \leq p \leq n$, where

   - $\mu_{i,j}(p, q)$ is the communication cost between tasks i and j when they are assigned to processors p and q respectively, and

   - $e_{i,p}$ is the execution cost when task i is assigned to processor p.

The first condition states that the communication cost between any two task instances (e.g. $t_{i,p}$ and $t_{j,q}$) is not negative. The second one depicts that for every edge $< i, j >$, the worst-case communication cost between any task instance $t_{i,p}$ and all its successor task instances (i.e. $t_{j,q}$'s, $\forall q$) is less than the minimum execution cost of task i.

2.3 Communication Model

The communication model we considered is different from that of reliability-oriented replication. In reliability-oriented replication problem, the objective is to increase the degree of fault tolerance. To detect fault and maintain data consistency, each task has to receive multiple copies of data from several task instances if its predecessor is replicated in more than one place.

The purpose of the replication problem considered in this paper is to decrease the sum of execution and communication costs. Under such consideration, there is no need to enforce plural communication between any two task instances. Hence, we propose the *1-out-of-n* communication model. In the model, for each edge $< i, j > \in E$, a task instance $t_{j,q}$ may start its execution if it receives the data from *any one* task instance of its predecessor, task i.

3   Problem Formulation and Complexity

Based on the computational model presented in Section 2.2, the problem of minimizing the total sum of execution and communication costs for an SP task graph can be approached by replication of tasks. An example where the replication may lead to a lower sum of execution costs and communication costs is given in Figure 2, where the number of processors in the system is two, and the execution costs and communication costs are listed in e table and μ table respectively. If each task is allowed to run on at most one processor, then the optimal allocation will be to assign task a to processor 1, b to 1, c to 1, d to 2, e to 2, and f to 1. The minimum cost is 68. However, if each task is allowed to be replicated more than one copies, (i.e. to replicate task a to processors 1 and 2), then the cost is 67.

We introduce integer variable $X_{i,p}$'s, $\forall\ 1 \leq i \leq M$ and $1 \leq p \leq n$, to formulate the problem where each $X_{i,p} = 1$ if task i is replicated on processor p; and $= 0$, otherwise. We define a binary function $\delta(x)$. If $x > 0$ then $\delta(x) = 1$ else $\delta(x) = 0$. We also associate an *allocated flag* $F(w)$ with each node w in the parsing tree, where $F(w) = 1$ if the allocation for tasks in the subtree S_w is valid; and $= 0$, otherwise. A valid allocation for the tasks in S_w is an allocation that follows the semantics of T_{chain}, T_{and}, and T_{or} subgraphs. A valid allocation is not necessarily the allocation in which each task in S_w is allocated to at least one processor. Some tasks in T_{or} subgraphs may be neglected without effecting the successful execution of an SP graph.

Given an SP graph G, its parsing tree $T(G)$ and any internal node w in $T(G)$, allocated flag $F(w)$ can be recursively computed:

1. if w is a T_{unit} node with a child i, then

$$F(w) = F(i) = \delta(\sum_{p=1}^{n} X_{i,p})$$

2. if w is a T_{chain} node with c children, $F(w) = F(child_1) \times F(child_2) \times \ldots \times F(child_c)$.

3. if w is a T_{and} node with forker s, joiner t and c children, then $F(w) = F(s) \times F(t) \times F(child_1) \times F(child_2) \times \ldots \times F(child_c)$.

4. if w is a T_{or} node with forker s, joiner t and c children, then $F(w) = F(s) \times F(t) \times \delta(F(child_1) + F(child_2) + \ldots + F(child_c))$.

The minimum cost replication problem for SP graphs, MCRP-SP, can be formulated as 0-1 integer programming problem, i.e:

$$Z = \text{Minimize} \left[\sum_{i,p} X_{i,p} * e_{i,p} + \sum_{<i,j> \in E, \, 1 \le q \le n} \min_{X_{i,p}=1} (\mu_{i,j}(p,q) * X_{j,q}) \right]$$

subject to $F(r) = 1$, where r is the root of $T(G)$ and $X_{i,p} = 0$ or $1, \forall i, p$.   (1)

The restricted problem which allows each task to run on at most one processor has the following formulation.

$$Z = \text{Minimize} \left[\sum_{i,p} X_{i,p} * e_{i,p} + \sum_{<i,j> \in E, p, q} \mu_{i,j} * X_{i,p} * X_{j,q} \right]$$

subject to $\sum_{p=1}^{n} X_{i,p} \le 1$ and $F(r) = 1$,

where r is the root of $T(G)$ and $X_{i,p} = 0$ or $1, \forall i, p$.   (2)

The task assignment problem (2) for SP graphs of M tasks onto n processors, has been solved in $O(n^3 M)$ time [12]. However, the multiprocessor task assignment for general types of task graphs without replication has been reported to be NP-complete [9]. As for the MCRP-SP problem, it can be shown to be NP-complete. In this paper, we are able to solve the problem and present a linear-time algorithm that is linear in the number of tasks when the number of processors is fixed for computation-intensive applications.

8

3.1 Assignment Graph

Bokhari [1] introduced the assignment graph to solve the task assignment problem (2). To prove the NP completeness of problem (1) and solve the problem, we also adopt the concept of the assignment graph of an SP graph. The assignment graph of an SP graph can be defined similarly. The following definitions apply to the assignment graph. And we draw up an assignment graph for an SP graph in Figure 3.

1. It is a directed graph with weighted nodes and edges.

2. It has $M \times n$ nodes. Each weighted node is labeled with a task instance, $t_{i,p}$.

3. A *layer* i is the collection of n weighted nodes ($t_{i,1}$, $t_{i,2}$, ..., and $t_{i,n}$). Each layer of the graph corresponds to a node in the SP graph. The layer corresponding to the source (sink) is called source (sink) layer.

4. A part of the assignment graph corresponds to an SP subgraph of type T_{chain}, T_{and} or T_{or} is called a T_{chain}, T_{and} or T_{or} *limb* respectively.

5. Communication costs are accounted for by giving the weight $\mu_{i,j}(p, q)$ to the edge going from $t_{i,p}$ to $t_{j,q}$.

6. Execution costs are assigned to the corresponding weighted nodes.

Given an assignment graph, Bokhari [1] solves Problem (2) by selecting one weighted node from each layer and including the weighted edges between any two selected nodes. This resulting subgraph is called an *allocation graph*. To solve Problem (1), more than one weighted node from each layer may be chosen. Similarly, a *replication graph* for Problem (1) can be constructed from an assignment graph by including all selected nodes and edges between these nodes. Examples of an allocation graph and a replication graph are shown in Figure 4 for an assignment graph shown in Figure 3. Note that for each node x in the replication graph there is only one edge incident to it from each predecessor layer of x.

In a replication graph, each layer may have more than one selected node. Let Variable $\bar{X}_l$ = $(X_{l,1}, X_{l,2}, ..., X_{l,n})$ be a replication vector for layer l in a replication graph. We define the

minimum *activation cost* of vector $\bar{X}_i$ for layer i, $A_i(\bar{X}_i)$, to be the minimum sum of the weights of all possible nodes and edges leading to the selected nodes of layer i in a replication graph. Then the goal of Problem (1) can be achieved by computing the minimal value of $\{A_{sink}(\bar{X}_{sink}) + \sum_{p=1}^{n} X_{sink,p} * e_{sink,p}\}$ over all possible values of $\bar{X}_{sink}$.

3.2   Complexity

In this section, we can show that Problem (1) for a computation-intensive application is NP-complete provided we prove the following:

Lemma 1: For any layer l in the replication graph, the minimum activation cost for two selected nodes $t_{l,p}$ and $t_{l,q}$ will be always greater than that for either node $t_{l,p}$ or $t_{l,q}$ only.

Proof: The Lemma can be proven by contradiction. Let A_1 be the the minimum activation cost for two nodes $t_{l,p}$ and $t_{l,q}$, and A_2 and A_3 be the minimum costs for $t_{l,p}$ and $t_{l,q}$ respectively. Assume that $A_1 < A_2$ and $A_1 < A_3$. Since A_1 includes the activation cost of node $t_{l,p}$, an activation cost for $t_{l,p}$ only can be obtained from A_1. The obtained value c is not necessarily the minimum value for $t_{l,p}$, hence $A_2 \leq c$. The value c is obtained by removing some weighted nodes and edges from replication graph. This implies that $c < A_1$. From above, we find that $A_2 < A_1$, which contradicts the assumption. The same reasoning can be applied to A_3 and reaches a contradiction. Therefore, the assumptions are incorrect and Lemma 1 holds.

□

Lemma 1 can be further extended to the cases where more than two weighted nodes are chosen. The conclusion we can draw is that the more nodes are selected from a layer, the bigger the activation cost is.

Lemma 2: Given a computation-intensive application with its SP task graph $G = (V, E)$ and its assignment graph, if node i has outdegree one and edge $< i, j > \in E$, then for any vector $\bar{X}_j$, the minimal activation cost $A_j(\bar{X}_j)$ can be obtained by choosing only one weighted node from layer i. (i.e. $\sum_{p=1}^{n} X_{i,p} = 1$)

Proof: The Lemma can be proven by contradiction. Since node i has outdegree one and edge

$$< \quad A_i(\bar{X}_i^0) + \sum_{p=1}^n X_{i,p}^0 * e_{i,p} + \sum_{q=1}^n \min_{X_{i,p}^0=1} (X_{j,q} * \mu_{i,j}(p,q)) = m.$$

The result, $m' < m$, contradicts our assumption. It means that the assumption is wrong and Lemma 2 holds.

□

Lemma 3: Given a computation-intensive application with its SP task graph G, the objective of the minimum cost can be achieved by considering only the replication of the forkers.

Proof: We proceed to prove the lemma by contradiction. Let the minimum cost for task replication problem be z_0 if only the forkers(i.e. outdegree > 1) are allowed to run on more than one processor. Assume the total cost can be reduced further by replicating some task i which is not a forker. Then there are two possible cases for i:

1. i has outdegree 0.

2. i has outdegree 1.

In case 1, i is the sink of the whole graph. Also i may be the joiner of some SP subgraphs. If i is allowed to run on an extra processor b, which is different from the one which i is initially assigned to (when z_0 is obtained), then the new cost will be $z_0 + e_{i,b} + \sum_{<d,i>\in E} \mu_{d,i}$. Apparently, the new cost is greater than z_0. This contradicts our assumption that the total cost can be reduced further by replicating task i.

In case 2, i has one successor. Let $< i,j > \in E$. From the assumption, we know that the replication of i can reduce the total cost. Hence, the minimum activation cost for task instances in layer j, $A_j(\bar{X}_j)$, is obtained when task i is replicated onto more than one processor. This contradicts Lemma 2. Hence, the assumption is incorrect and the objective of the minimum cost can be achieved by considering only the replication of the forkers.

□

Lemma 3 tells that, given an SP graph, if we can find out the optimal replication for the forkers, Problem (1) for computation-intensive applications can be solved. Now, we show that the problem

$< i, j > \in E$, we know that

$$A_j(\bar{X}_j) = \min_{\bar{X}_i} \{ A_i(\bar{X}_i) + \sum_{p=1}^{n} X_{i,p} * e_{i,p} + \sum_{q=1}^{n} \min_{X_{i,p}=1} (X_{j,q} * \mu_{i,j}(p,q)) \}.$$

Let us assume that the above equation reaches a minimal value m when more than one node from layer i is selected and the optimal replication vector is $\bar{X}_i^0$. Since $\sum_{p=1}^{n} X_{i,p} > 1$ for $\bar{X}_i^0$, we may remove one selected node from layer i and obtain a new vector $\bar{X}_i'$. Without loss of generality, let us remove $t_{i,r}$. By removing node $t_{i,r}$, a new value m' is obtained. Since m is the minimum value for layer i, it implies that $m \leq m'$.

From Lemma 1, we obtain that $A_i(\bar{X}_i') < A_i(\bar{X}_i^0)$. And for a computation-intensive application, the following holds that $\sum_{q=1}^{n} \mu_{i,j}(p,q) \leq \min_p(e_{i,p})$, $\forall\ 1 \leq p \leq n$. Then,

$$
\begin{aligned}
m' &= A_i(\bar{X}_i') + \sum_{p=1}^{n} X_{i,p}' * e_{i,p} + \sum_{q=1}^{n} \min_{X_{i,p}'=1} (X_{j,q} * \mu_{i,j}(p,q)) \\
&< A_i(\bar{X}_i^0) + \sum_{p=1}^{n} X_{i,p}' * e_{i,p} + \sum_{q=1}^{n} \min_{X_{i,p}'=1} (X_{j,q} * \mu_{i,j}(p,q)) \\
&< A_i(\bar{X}_i^0) + (\sum_{p=1}^{n} X_{i,p}^0 * e_{i,p} - e_{i,r}) + \sum_{q=1}^{n} \min_{X_{i,r}'=1} (X_{j,q} * \mu_{i,j}(p,q)) \\
&= A_i(\bar{X}_i^0) + \sum_{p=1}^{n} X_{i,p}^0 * e_{i,p} + [\sum_{q=1}^{n} \min_{X_{i,p}'=1} (X_{j,q} * \mu_{i,j}(p,q))] - e_{i,r} \\
&< A_i(\bar{X}_i^0) + \sum_{p=1}^{n} X_{i,p}^0 * e_{i,p} + [\sum_{q=1}^{n} \min_{X_{i,p}'=1} (X_{j,q} * \mu_{i,j}(p,q))] - \min_p(e_{i,p}) \\
&\leq A_i(\bar{X}_i^0) + \sum_{p=1}^{n} X_{i,p}^0 * e_{i,p} + [\sum_{q=1}^{n} \min_{X_{i,p}'=1} (X_{j,q} * \mu_{i,j}(p,q))] - \sum_{q=1}^{n} \mu_{i,j}(p,q) \\
&< A_i(\bar{X}_i^0) + \sum_{p=1}^{n} X_{i,p}^0 * e_{i,p}
\end{aligned}
$$

$$< \quad A_i(\bar{X}_i^0) + \sum_{p=1}^{n} X_{i,p}^0 * e_{i,p} + \sum_{q=1}^{n} \min_{X_{i,p}^0=1} (X_{j,q} * \mu_{i,j}(p,q)) = m.$$

The result, $m' < m$, contradicts our assumption. It means that the assumption is wrong and Lemma 2 holds.

□

Lemma 3: Given a computation-intensive application with its SP task graph G, the objective of the minimum cost can be achieved by considering only the replication of the forkers.

Proof: We proceed to prove the lemma by contradiction. Let the minimum cost for task replication problem be z_0 if only the forkers(i.e. outdegree > 1) are allowed to run on more than one processor. Assume the total cost can be reduced further by replicating some task i which is not a forker. Then there are two possible cases for i:

1. i has outdegree 0.

2. i has outdegree 1.

In case 1, i is the sink of the whole graph. Also i may be the joiner of some SP subgraphs. If i is allowed to run on an extra processor b, which is different from the one which i is initially assigned to (when z_0 is obtained), then the new cost will be $z_0 + e_{i,b} + \sum_{<d,i> \in E} \mu_{d,i}$. Apparently, the new cost is greater than z_0. This contradicts our assumption that the total cost can be reduced further by replicating task i.

In case 2, i has one successor. Let $< i,j > \in E$. From the assumption, we know that the replication of i can reduce the total cost. Hence, the minimum activation cost for task instances in layer j, $A_j(\bar{X}_j)$, is obtained when task i is replicated onto more than one processor. This contradicts Lemma 2. Hence, the assumption is incorrect and the objective of the minimum cost can be achieved by considering only the replication of the forkers.

□

Lemma 3 tells that, given an SP graph, if we can find out the optimal replication for the forkers, Problem (1) for computation-intensive applications can be solved. Now, we show that the problem

13

of finding an optimal replication for the forkers in an SP graph is NP-complete. First, a special form of the replication problem is introduced.

Uni-Cost Task Replication (UCTR) problem is stated as follows:

INSTANCE: Graph $G' = (V', E')$, $V' = V'_1 \cup V'_2$, where $|V'_1| = n$ and $|V'_2| = m$. If $x \in V'_1$ and $y \in V'_2$ then edge $<x, y> \in E'$ (i.e. $|E'| = m \times n$). For each $x \in V'_1$, there is an activation cost m. Associated with each edge $<x, y> \in E'$, there is a communication cost $d_{x,y} = n \times m$ or 0. A positive integer $K \leq n \times m$ is also given.

QUESTION: Is there a feasible subset $V_k \subseteq V'_1$ such that, we have

$$[\sum_{x \in V_k} m + \sum_{y \in V'_2} \min_{x \in V_k}(d_{x,y})] \leq K? \tag{3}$$

[Theorem 1]: Uni-Cost Task Replication problem is NP-Complete.

[Proof]: The problem is in NP because a subset V_k, if it exists, can be checked to see if the sum of activation costs and communication costs is less than or equal to K. We shall now transform the VERTEX COVER [3] problem to this problem. Given any graph $G = (V, E)$ and an integer $C \leq |V|$, we shall construct a new graph $G' = (V', E')$ and $V' = V'_1 \cup V'_2$, such that there exists a VERTEX COVER of size C or less in G if and only if there is a feasible subset of V'_1 in G'. Let $|V| = n$ and $|E| = m$. To construct G', (1) we create a vertex v_i for each node in V, (2) we number the edges in E, and (3) we create a vertex b_j for each edge $<u, v> \in E$ where $u, v \in V$. We define $K = m \times C$, $V'_1 = \{v_1, v_2, \ldots, v_n\}$, $V'_2 = \{b_1, b_2, \ldots, b_m\}$ and $E' = \{<v_x, b_y> \mid v_x \in V'_1, b_y \in V'_2\}$. Let $d_{v_x, b_y} = 0$, if v_x is an end point of the corresponding edge of vertex b_y; and $= n \times m$, otherwise. An illustration, where $n = 7$ and $m = 9$, is shown in Figure 5.

Let us now argue that there exists a vertex cover of size C or less in G if and only if there is a feasible subset of V'_1 in G' to satisfy that the sum of activation cost and communication cost is $m \times C$ or less. Suppose there is a vertex cover of size C, then for each vertex b_y ($= <u, v>$) in V'_2, at least one of u and v belongs to the vertex cover. By selecting all the vertices in the vertex cover into the subset of V'_1, we know that the sum in Eq. (3) will be $m \times C$. Since $C \leq n$, it implies that $m \times C \leq n \times m$.

Conversely, for any feasible subset $V_k \subseteq V'_1$ such that the total cost is equal to or less than

14

mC, we can see that the second term of Eq. (3) (i.e. the sum of communication cost) must be zero. Suppose, for some $g_y \in V'_2$, the minimum communication cost between g_y and vertices in V' is nonzero, then the communication cost will be at least $m \times n$. Since $C \leq n$, it implies that $m \times n \geq m \times C$. The total cost in Eq. (3) will be greater than $m \times C$, which is a contradiction. Thus the minimum communication cost between any vertex in V'_2 and any vertex in V_k is zero. It means that at least one of two end points of each edge in E belongs to V_k. Since, there is at most C vertices in V_k (the activation cost for each vertex is m), and by selecting the vertices in V_k, we obtain a vertex cover of size C or less in G.

<div align="right">□</div>

[Theorem 2]: *The problem, MCRP-SP for computation-intensive applications, is NP-complete.*
[Proof]: From Lemma 3, we know that only the forker in an SP graph of type T_{and} needs to run on more than one processor. Consider the following recognition version of Problem (1) for SP graphs of type T_{and}:

Given a distributed system of n processors, an SP graph $G^a = (V^a, E^a)$ of type T_{and}, its assignment graph H and two positive integers m and r. Let r be a multiple of m, $V^a = \{s, t, 1, 2, \ldots, r\}$ and $E^a = \{< s, i > \mid i = 1, 2, \ldots, r\} \cup \{< i, t > \mid i = 1, 2, \ldots, r\}$. Task s (t) is the forker (joiner) of G^a. Execution cost $e_{i,p}$ and communication cost $\mu_{i,j}(p, q)$ are defined in H, $\forall < i, j > \in E^a$ and $\forall 1 \leq p, q \leq n$. Integer variable $X_{i,p} = 1$ if task i is assigned to processor p; and $= 0$, otherwise. When a positive integer $K \leq r$ is given, is there an assignment of $X_{i,p}$'s, such that

$$[\sum_{i,p} X_{i,p} * e_{i,p} + \sum_{<i,j> \in E, \ 1 \leq q \leq n} \min_{X_{i,p}=1} (\mu_{i,j}(p,q) * X_{j,q})] \leq K?$$

$$\text{where} \sum_{i,p} X_{i,p} = 1, \ \forall i \neq s, \text{ and } \sum_{i,p} X_{i,p} \geq 1, \text{ if } i = s. \tag{4}$$

We shall transform the UCTR problem to this problem. Given any graph $G' = (V'_1 \cup V'_2, E')$ considered in UCTR problem, we construct an SP graph of type T_{and}, $G^a = (V^a, E^a)$, and its assignment graph H, such that G' has a feasible subset of V'_1 to allow the sum in Eq. (3) is K or less if and only if there is an assignment of $X_{i,p}$'s for G^a and H to satisfy Eq. (4). Let $\mid V'_1 \mid = n$,

<div align="center">15</div>

$|V_2'| = m$, then the unit cost $f = n \times m$. Assign $r = m \times f$ $(= n \times m^2)$ and $K = n \times m$. The forker and joiner of G^a are s and t respectively. Then $V^a = \{s, t, 1, 2, \ldots, r\}$ and $E^a = \{< s, i > \mid i = 1, 2, \ldots, r\} \cup \{< i, t > \mid i = 1, 2, \ldots, r\}$. We assign the execution costs and communication costs in H as follows. An illustration, where $m = 2$ and $n = 3$, is shown in Figure 7.

- $\forall\ 1 \leq p \leq n, e_{s,p} = m$.

- $\forall\ 1 \leq i \leq r, \forall\ 1 \leq p \leq n$, if $p = 1$ then $e_{i,p} = 0$ else $e_{i,p} = r$.

- $\forall\ 1 \leq p \leq n$, if $p = 1$ then $e_{t,p} = 0$ else $e_{t,p} = r$.

- $\forall\ 1 \leq i \leq r, \forall\ 1 \leq p \leq n$, let $q = (i - 1)$ div $(m \times n)$, where div is the integral division. If $d_{v_p, b_{q+1}} \neq 0$ then $\mu_{s,i}(p, 1) = 1$ else $\mu_{s,i}(p, 1) = 0$.

- $\forall\ 1 \leq i \leq r, \forall\ 1 \leq p \leq n, \forall\ q \neq 1, \mu_{s,i}(p, q) = 0$.

- $\forall\ 1 \leq i \leq r, \forall\ 1 \leq p, q \leq n, \mu_{i,t}(p, q) = 0$.

It is easy to verify that the SP graph constructed by the the above rules is of type T_{and} and computation-intensive. For each node in V_2' of G', we create f nodes in G^a, where the communication cost between each node and source s is either one or zero.

Let us now argue that there exists a feasible subset of V_1' for UCTR problem if and only if there exists a valid assignment of $X_{i,p}$'s such that the total sum in Eq. (4) is K or less. Suppose a feasible subset V_k of V_1' exists such that the sum in Eq. (3) is $C\ (\leq K)$. Let V_1' be $\{v_1, v_2, \ldots, v_n\}$ Then we can obtain a valid assignment by letting $X_{i,1} = 1, X_{i,2} = 0, \ldots, X_{i,n} = 0, \forall\ 1 \leq i \leq r$, and $X_{t,1} = 1, X_{t,2} = 0, \ldots, X_{t,n} = 0$, and $X_{s,p} = 1$, if $v_p \in V_k$; and $X_{s,p} = 0$, if $v_p \notin V_k, \forall\ 1 \leq p \leq n$. Since each node x in V_2' corresponds to f nodes in G^a, it is sure that the communication cost between node x and any node (v_p) in V_1' is equal to the total communication costs between these f nodes and any task instance of source $(t_{s,p})$ in G^a. By summing up all the costs, we can obtain that the total sum is C. Since $C \leq K \leq n \times m < r$, this is a valid assignment.

Conversely, if there exists an assignment of $X_{i,p}$'s such that the sum in Eq. (4) is K or less, then the following must be true that $X_{i,1} = 1, X_{i,2} = 0, \ldots, X_{i,n} = 0, \forall\ 1 \leq i \leq r$, and $X_{t,1} = 1, X_{t,2} = 0, \ldots, X_{t,n} = 0$. It is because for some $p \neq 1$, if $X_{i,p} = 1$ then the sum must be greater than

r, which causes a conflict. Hence the second term in Eq. (4) must be zero. Thus, we may obtain a subset of V_1 for UCTR problem by selecting node $x \in V_1$ if $X_{s,x}$ equals 1. Since the first term in Eq. (3) is equivalent to the first term in Eq. (4), the total sum for UCTR problem will be also K or less then.

□

4   Optimal Replication for SP Graphs of Type T_{and}

In this section, we develop the branch-and-bound algorithm to find an optimal solution for T_{and} subgraphs. The non-forker nodes only need to run on one processor. Hence, an optimal assignment of non-forker nodes can be done after an optimal replication for forkers is obtained.

4.1   A Branch-and-Bound Method for Optimal Replication

Consider a T_{and} SP graph with forker-joiner pair (s,h) shown in Figure 6. There are B subgraphs connected by s and h. These B subgraphs have a parallel-and relationship. Since the joiner h has only one copy in optimal solution (i.e. $\sum_{p=1}^{n} X_{h,p} = 1$), we decompose the minimum cost replication problem $\mathcal{P}$ for a T_{and} SP graph into n subproblems $\mathcal{P}^q$, $q = 1, 2, \ldots, n$, where $\mathcal{P}^q$ is to find the minimum cost when the joiner is assigned to processor q (i.e. $X_{h,q} = 1$).

Given a joiner instance $t_{h,q}$, subgraphs G_b's, $b = 1, 2, \ldots, B$, and the minimum costs $C_{p,q}^b$'s between each forker instance $t_{s,p}$ and joiner instance $t_{h,q}$, $\forall\ 1 \leq p \leq n$ and $1 \leq b \leq B$. we further decompose problem $\mathcal{P}^q$ into n subproblems $\mathcal{P}_k^q$, $k = 1, 2, \ldots, n$, where k is the number of replicated copies that the forker s has. Basically, $\mathcal{P}_k^q$ means the problem of finding an optimal replication for k copies of forker s where the joiner h is assigned to processor q. Since the problem of finding an optimal replication for forker s is NP-complete, we propose a branch-and-bound algorithm for each subproblem $\mathcal{P}_k^q$.

We sort the forker instances according to their execution costs $e_{s,p}$'s into non-decreasing order. Without loss of generality, we assume $e_{s,1} \leq e_{s,2} \leq \ldots \leq e_{s,n}$. We represent all the possible combinations that s may be replicated by a combination tree with $\binom{n}{k}$ leaf nodes. To make the solution efficient, we shall not consider all combinations since it is time-consuming. We apply a

least-cost branch-and-bound algorithm to find an optimal solution by traversing a small portion of the combination tree.

During the search, we maintain a variable $\hat{z}$ to record the minimum value known so far. The search is done by the expansion of intermediate nodes. Each intermediate node v at level y represents a combination of y out of n forker instances. The expansion of node v generates at most $n - y$ child nodes, while each child node inherits y forker instances from v and adds one distinct forker instance to itself. For example, if node v is represented by $\prec t_{s,i_1}, t_{s,i_2}, \ldots, t_{s,i_y} \succ$, where $i_1 < i_2 < \ldots < i_y$, then $\prec t_{s,i_1}, t_{s,i_2}, \ldots, t_{s,i_y}, t_{s,i_y+j} \succ$ represents a possible child node of v, $\forall\, 1 \leq j \leq n - i_y$. A combination tree, where $k = 4$ and $n = 6$, is shown in Figure 8. At any intermediate node of a combination tree, we apply an estimation function to compute the least cost this node can achieve. If the estimated cost is greater than $\hat{z}$, then we prune the node and the further expansion of the node is not necessary. Otherwise, we insert this node along with its estimated cost into a queue. The nodes in the queue are sorted into non-decreasing order of their estimated costs, where the first node of the queue is always the next one to be expanded. When the expansion reaches a leaf node, the actual cost of this leaf is computed. If the cost is less than $\hat{z}$, we update $\hat{z}$. The algorithm terminates when the queue is empty.

4.1.1 The Estimation Function

The proposed branch-and-bound algorithm is characterized by the estimation function. Let node v be at level y of the combination tree associated with subproblem $\mathcal{P}_k^q$ and be represented by $\prec t_{s,i_1}, t_{s,i_2}, \ldots, t_{s,i_y} \succ$, where $i_1 < i_2 < \ldots < i_y$. Any leaf node that can be reached from node v needs $k - y$ more forker instances. Let $\ell = \prec j_1, j_2, \ldots, j_{k-y} \succ$ be a tuple of $k - y$ instances chosen from the remaining $n - i_y$ instances, where $j_1 < j_2 < \ldots < j_{k-y}$. Let L be the set of all possible ℓ's. Let $g(v)$ be the smallest cost among all leaf nodes that can be reached from node v.

$$g(v) = \sum_{a=1}^{y} e_{s,i_a} \;+\; \min_{\ell \in L} [\sum_{j_x \in \ell} e_{s,j_x} \;+\; \sum_{b=1}^{B} \min_{p = i_1, i_2, \ldots, i_y \text{ or } p \in \ell} (C_{p,q}^b)\,] \;+\; e_{h,q}.$$

Since the complexity involved in computing $g(v)$ is $\binom{n-i_y}{k-y}$, we use the following estimation function $est(v)$ to approximate $g(v)$:

$$est(v) = \sum_{a=1}^{y} e_{s,i_a} + \sum_{j=i_y+1}^{i_y+k-y} e_{s,j} + \sum_{b=1}^{B} \min_{p=i_1,i_2,\dots,i_y,i_y+1,i_y+2,\dots,n} (C_{p,q}^b) + e_{h,q}. \tag{5}$$

Since

$$\sum_{j=i_y+1}^{i_y+k-y} e_{s,j} \leq \sum_{j_x \in \ell} e_{s,j_x} \quad \text{and} \quad \sum_{b=1}^{B} \min_{p=i_y+1,i_y+2,\dots,n} (C_{p,q}^b) \leq \sum_{b=1}^{B} \min_{p \in \ell} (C_{p,q}^b) \,,$$

it is easy to see that $est(v) \leq g(v)$. Hence, we use $est(v)$ as the lower bound of the objective function at node v.

4.1.2 The Proposed Algorithm

Three parameters of the branch-and-bound algorithm are joiner instance ($t_{h,q}$), the number of processors that forker s is allowed to run (k), and the up-to-date minimum cost ($\hat{z}$). The algorithm $BB(k, q, \hat{z})$ is shown in Table 1.

The MCRP-SP problem can be solved by invoking $BB(k, q, \hat{z})$ n^2 times with parameters set to different values. $BB(k, q, \hat{z})$ solves the problem $\mathcal{P}_k^q$, while the whole procedure, shown in Table 2, solves $\mathcal{P}$.

4.2 Performance Evaluation

The essence of the branch-and-bound algorithm is the expansion of the intermediate nodes. Upon the removal of a node from the queue its children are generated and their estimated values are computed. If the estimation function performs well and gives a tight lower bound of objective function, the number of expanded nodes should be small. Then an optimal solution can be found out as soon as possible.

We conduct two sets of experiments to evaluate the performance of the proposed solution. The performance indices we consider are the number of enqueued intermediate nodes (EIM) and the number of visited leaf nodes (VLF) during the search. We calculate EIM and VLF by inserting one

19

counter for each index at lines 13 and 8 of Table 1 respectively. Each time the execution reaches line 13 (8), EIM (VLF) is incremented by 1.

The first set of experiments is on SP graphs of type T_{and} where the communication cost between any two task instances is arbitrary and is generated by random number generator within the range [1,50]. The execution cost for each task instance is also randomly generated within the same range. The second set of experiments is on SP graphs of type T_{and} with the constrain of computation-intensive applications. We vary the size of the problem by assigning different values to the number of processors in the system (n) and the number of parallel-and subgraphs connected by forker and joiner (B). For each size of the problem (n, B), we randomly generate 50 problem instances and solve them. The results, including the average values of EIM and VLF over the solutions of 50 problem instances, are summarized in Table 3.

From Table 3, we find out that the proposed method significantly reduces the number of expansions for intermediate nodes and leaf nodes. For example, for problem size $(n, B) = (20, 40)$, the total number of leaf nodes is 2^{20} (= 1,048,576) if an exhaustive search is applied. However, our algorithm only generates 16,857 nodes on the average, because we apply $est(v)$, $\hat{z}$, and the branch-and-bound approach.

The branch-and-bound approach and the estimation function even perform better for the computation-intensive applications. We can see that EIM and VLF values are much more smaller in Set II than those in Set I. It is because that in the computation-intensive applications an optimal number of replications for the forker is smaller than that in general applications. The $\hat{z}$ value in function $OPT()$ is able to reflect this fact and avoid the unnecessary expansions.

5   Sub-Optimal Replication for SP Graphs of Type T_{and}

The branch-and-bound algorithm in section 4.1 yields an optimal solution for T_{and} subgraphs. However, the complexity involved is in exponential time in the worst case. Hence, we also consider to find a near-optimal solution in polynomial time.

5.1 Approximation Method

For the problem $\mathcal{P}_k^q$ defined in section 4.1, we exploit an approximation approach to solve it in polynomial time. The approach is based on iterative selection in a dynamic programming fashion. Given a joiner instance $t_{h,q}$ and subgraphs G_b, $b = 1, 2, \ldots, B$, and minimum costs $C_{p,q}^b$ between $t_{h,q}$ and $t_{s,p}$, $p = 1, 2, \ldots, n$, and $b = 1, 2, \ldots, B$. we define $Sub(p,b)$ to be the sub-optimal solution for replication of forker s where forker instances $t_{s,1}, t_{s,2}, \ldots, t_{s,p}$ and subgraphs $G_1, G_2, \ldots, G_b$ are taken into consideration.

Strategy 1:

$Sub(p,b)$ can be obtained from $Sub(p-1,b)$ by considering one more forker instance $t_{s,p}$. Strategy 1 consists of two steps. The first step is to initialize $Sub(p,b)$ to be $Sub(p-1,b)$ and to determine if $t_{s,p}$ is to be included into $Sub(p,b)$ or not. If yes, then add $t_{s,p}$ in. The second step is to examine if any instances in $Sub(p-1,b)$ should be removed or not. Due to the possible inclusion of $t_{s,p}$ in the first step, we may obtain a lower cost if we remove some instances $t_{s,i}$'s, $i < p$, and reassign the communications for some graphs G_j's from $t_{s,i}$'s to $t_{s,p}$.

Strategy 2:

$Sub(p,b)$ can also be obtained from $Sub(p,b-1)$ by taking one more subgraph G_b into account. Initially, $Sub(p,b)$ is set to be $Sub(p,b-1)$. The first step is to choose the best forker instance from $t_{s,1}, t_{s,2}, \ldots, t_{s,p}$ for G_b. Let the best instance be $t_{s,z}$. The second step is to see if $t_{s,z}$ is in $Sub(p,b)$ or not. If not, a condition is checked to decide whether $t_{s,z}$ should be added in or not. Upon the addition of $t_{s,z}$, we may remove some instances and reassign the communications to achieve a lower cost.

We compare two possible results obtained from the above two strategies and assign the one with lower cost to actual $Sub(p,b)$. Hence by computing in a dynamic programming fashion, $Sub(n,B)$ can be obtained. The algorithm and its graphical interpretation are shown in Figure 9.

5.2 Performance Evaluation

The complexity involved in each strategy described in section 5.1 is $O(nB)$. Since the solving of $Sub(n,B)$ needs to invoke $n \times B$ times of strategies 1 and 2, the total complexity of solving

21

$Sub(n, B)$ by the approximation method is $O(n^2 B^2)$.

We conduct a set of experiments to evaluate the performance of the approximation method. For each problem size (n, B), we randomly generate 50 instances and solve them by using approximation method and exhaustive searching. The data for computation and communication in the experiments are based on the uniform distribution over the range [1,50]. We compare the minimum cost obtained from exhaustive searching (EXHAUST) with those from from approximation (APPROX) and single assignment solution (SINGLE). The optimal single assignment solution is the one in which only one forker instance is allowed. Note that the solutions from SINGLE are obtained from the shortest path algorithm [1]. The results are summarized in Table 4. From the table, we find out that the approximation method yields a tight approximation of the minimum cost. On the contrary, the error range for single copy solution is at least 20%. This again justifies that the replication can lead to a lower cost than an optimal assignment does.

6 Solution of MCRP-SP for computation-intensive applications

6.1 The Solution

Given a computation-intensive application with its SP graph, we generate its parsing tree and assignment graph first. The algorithm finds the minimum weight replication graph from the assignment graph. Then the optimal solution is obtained from the minimum weight replication graph.

The algorithm traverses the parsing tree in the postfix order. Namely, during the traversal, an optimal solution of the subtree S_x, induced by an intermediate node x along with all x's descendant nodes, can be found only after the optimal solutions of x's descendant nodes are found. Given an SP graph G and a distributed system S, we know that there is a one-to-one correspondence between each subtree S_x in a parsing tree $T(G)$ and a limb in the assignment graph of G on S. Whenever a child node b of x is visited, the corresponding limb in the assignment graph will be replaced with a a two-layer T_{chain} limb if b is a T_{chain}- or T_{or}-type node; and a one-layer T_{unit} limb if b is a T_{and}-type node. The algorithm is shown in Table 5. A graphical demonstration of how the algorithm solves the problem is shown in Figure 10.

Before the replacement of a T_{chain} limb is performed (i.e. x is a T_{chain}-type node), each constituent child limb has been replaced with a T_{unit} or two-layer T_{chain} limb. Hence, the shortest

path algorithm [1] can be used to compute the weights of the new edges between each node in the source layer and each node in the sink layer of the new T_{chain} limb. The complexity, from lines 05 to 08 of Table 5, in transformation of the limb, corresponding to an intermediate node x with M children, into a two-layer T_{chain} limb is $O(Mn^3)$. An example of illustrating the replacement of a T_{chain} limb is shown from parts (b) to (c) and parts (d) to (e) in Figure 10.

For the replacement of a T_{and} limb, we have to compute $C^b_{p,q}$'s. The values can also be computed by the shortest path algorithm. Hence, the complexity involved in lines 16 and 17 is $O(Bn^3)$. According to the computational model in section 2.2, each task instance s may start its execution if it receives the necessary data from any task instance of its predecessor d. And, from Lemma 2, we know that the minimum sum of initialization costs of multiple task instances of s will be always from only one task instance of d. Therefore, the initialization of task instance $t_{s,p}$ depends on which task instance of d it communicates with. That is why, in line 19, the communication cost $\mu_{d,s}(i,p)$ is added to the the execution cost of $e_{s,p}$ before $OPT()$ is invoked. And the most significant part of the replacement is to compute the weights on the new edges from the source layer to sink layer. The complexity is $n^2 \times O(OPT())$, which in the worst case is $n^2 2^n$. However, in the average, our OPT function performs pretty well and reduces the complexity significantly. An example of illustrating the replacement of a T_{and} limb is shown from parts (c) to (d) in Figure 10.

We also consider to use the approximation method to find the sub-optimal replacement of a T_{and} limb. In that case, function $OPT()$ in line 21 is replaced with $Sub(n,B)$. The total complexity involved is $O(n^4 B^2)$ then.

Finally, for the replacement of a T_{or} limb, if there are B subgraphs connected between the forker and the joiner, then the complexity will be $O(Bn^2)$ for the new edges and $O(Bn^3)$ for $C^b_{p,q}$'s. An example of illustrating the replacement of a T_{or} limb is shown from parts (a) to (b) in Figure 10.

When the traversal reaches the root node of the parsing tree, the result of $FIND()$ will give us either one single layer or two layers, depending on the type of root node. All we have to do is to select the lightest of these n (in single layer) or n^2 (in two layers) shortest path combinations. An optimal replication graph itself is found by combining the shortest paths between the selected

23

nodes that were saved earlier. The whole algorithm has the complexity of

$$O(An^2 2^n) + \sum_i (R_i n^3) + \sum_i (C_i n^3)$$

where A is the number of T_{and} limbs, R_i is the number of subgraphs in the ith T_{or} limb, and C_i is the number of layers in the ith T_{chain} limb. This is not greater than $O(Mn^2 2^n)$, where M is the total number of tasks in the SP graph. The complexity of the algorithm is a linear function of M if the number of processors, n, is fixed.

6.2 Conclusion Remark

This paper has focused on MCRP-SP, the optimal replication problem of SP task graphs for computation-intensive applications. The purpose of replication is to reduce inter-processor communication, and to fully utilize the processor power in the distributed systems. The SP graph model, which is extensively used in modeling applications in distributed systems, is used. The applications considered in this paper are computation-intensive in which the execution cost of a task is greater than its communication cost. We prove that MCRP-SP is NP-complete. We present branch-and-bound and approximation methods for SP graphs of type T_{and}. The numerical results show that the algorithm performs very well and avoids a lot of unnecessary searching. Finally, we present an algorithm to solve the MCRP-SP problem for computation-intensive applications. The proposed optimal solution has the complexity of $O(n^2 2^n M)$ in the worst case, while the approximation solution is in the complexity of $O(n^4 M^2)$, where n is the number of processors in the system and M is the number of tasks in the graph.

For the applications in which the communication cost between two tasks is greater than the execution cost of a task, the replication can still be used to reduce the total cost. However, in the extreme case where the execution cost of each task is zero, the optimal allocation will be to assign each task to one processor. We are studying the optimal replication for the general case.

References

[1] S.H. Bokhari, *Assignment Problems in Parallel and Distributed Computing*, Kluwer Academic Publisheds, MA, 1987.

[2]   Y. Chen and T. Chen, "Implementing Fault-Tolerance via Modular Redundancy with Comparison," *IEEE Trans. Reliability*, Vol. 39, pp 217-225, June, 1990.

[3]   M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to Theory of NP-Completeness*, San Francisco: W.H. Freeman & Company, Publishers, 1979.

[4]   R. Jan, D. Liang and S.K. Tripathi, "A Linear-Time Algorithm for Computing Distributed Task Reliability in Pseudo Two-Terminal Series-Parallel Graphs," *submitted for publication to Journal of Parallel and Distributed Computing*.

[5]   C.C. Price and S. Krishnaprasad, "Software Allocation Models for Distributed Computing Systems," in *Proc. 4th International Conference on Distributed Computing Systems*, pp 40-48, May 1984.

[6]   D. Liang, A.K. Agrawala, D. Mosse, and Y. Shi, "Designing Fault Tolerant Applications in *Maruti*," *Proc. 3rd International Symposium on Software Reliability Engineering*, pp. 264-273, Research Triangle Park, NC, Oct. 1992.

[7]   V.M. Lo, "Heuristic Algorithms for Task Assignment in Distributed Systems," in *Proc. 4th International Conference on Distributed Computing Systems*, pp 30-39, May 1984.

[8]   P.R. Ma, E.Y.S. Lee and M. Tsuchiya, "A Task Allocation Model for Distributed Computing Systems," *IEEE Trans. Computers*, Vol. C-31, pp 41-47, Jan. 1982.

[9]   V.F. Magirou and J.Z. Milis, "An Algorithm for the Multiprocessor Assignment Problem," *Operations Research Letters*, Vol. 8, pp 351-356, Dec. 1989.

[10] C.C. Price and U.W. Pooch, "Search Techniques for a Nonlinear Multiprocessor Scheduling Problem," *Naval Res. Logist. Quart.*, Vol 29, pp 213-233, June 1982.

[11] H.S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithm," *IEEE Trans. Soft. Eng.* Vol 3, pp 85-93, Jan. 1977.

[12] D. Towsley, "Allocating Programs Containing Branches and Loops Within a Multiple Processor System," *IEEE Trans. Software Eng.*, Vol. SE-12, pp. 1018-1024, Oct, 1986.
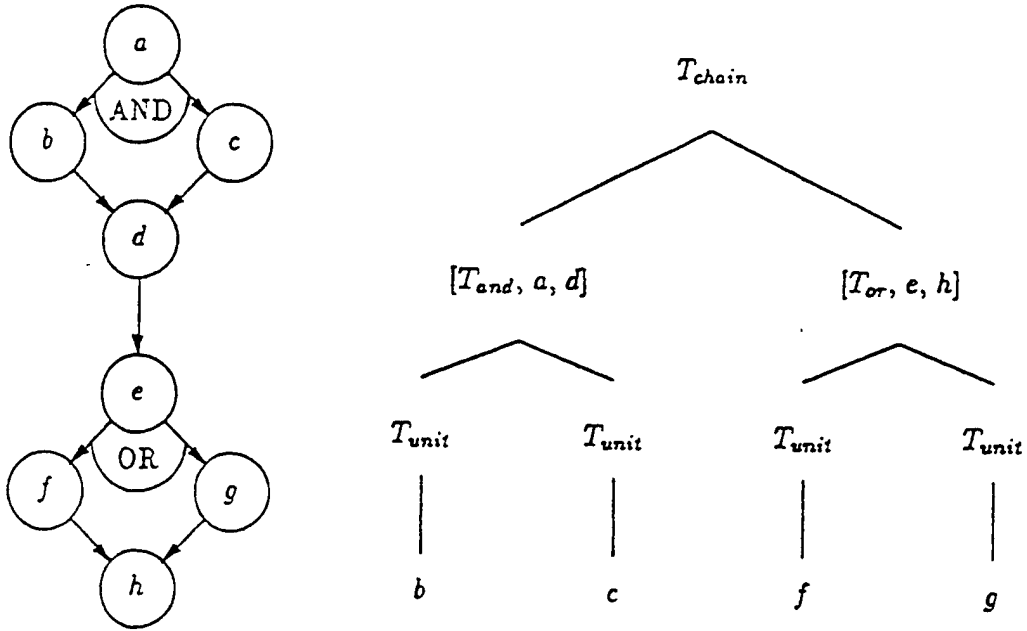
Figure 1: An SP graph and its parsing tree

e table	processor 1	processor 2
task a on	5	5
task b on	7	16
task c on	10	20
task d on	25	8
task e on	14	6
task f on	10	13

μ table	$t_{b,1}$	$t_{b,2}$	$t_{c,1}$	$t_{c,2}$	$t_{d,1}$	$t_{d,2}$	$t_{e,1}$	$t_{e,2}$
$t_{a,1}$ to	1	4	1	4	1	4	1	4
$t_{a,2}$ to	4	1	4	1	4	1	4	1
to $t_{f,1}$ from	3	3	3	3	3	3	3	3
to $t_{f,2}$ from	3	3	3	3	3	3	3	3
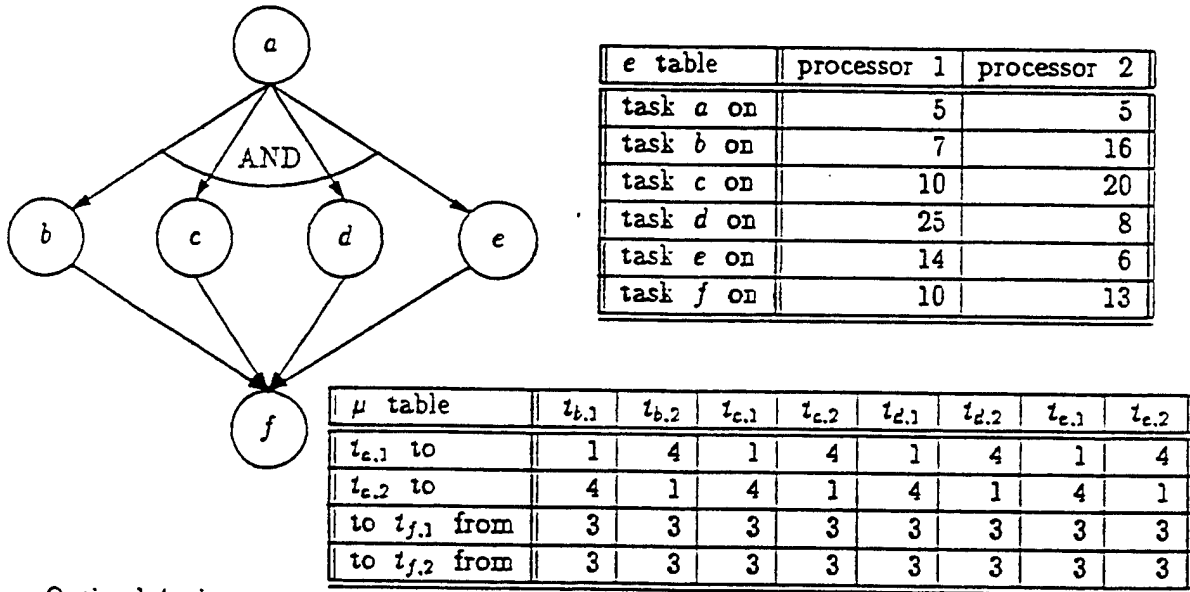
Optimal Assignment:

$$e_{a,1} + \mu_{a,b}(1,1) + \mu_{a,c}(1,1) + \mu_{a,d}(1,2) + \mu_{a,e}(1,2) + e_{b,1} + e_{c,1}$$
$$+ e_{d,2} + e_{e,2} + \mu_{b,f}(1,1) + \mu_{c,f}(1,1) + \mu_{d,f}(2,1) + \mu_{e,f}(2,1) + e_{f,1} = 68$$

Optimal Replication:

$$e_{a,1} + e_{a,2} + \mu_{a,b}(1,1) + \mu_{a,c}(1,1) + \mu_{a,d}(2,2) + \mu_{a,e}(2,2) + e_{b,1}$$
$$+ e_{c,1} + e_{d,2} + e_{e,2} + \mu_{b,f}(1,1) + \mu_{c,f}(1,1) + \mu_{d,f}(2,1) + \mu_{e,f}(2,1) + e_{f,1} = 67$$

Figure 2: An example to show how the replication can reduce the total cost
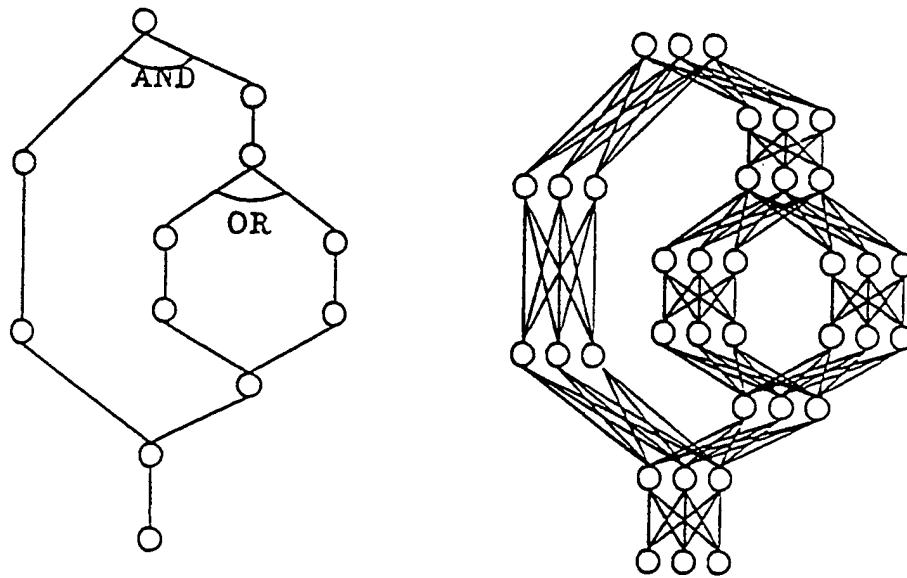
26

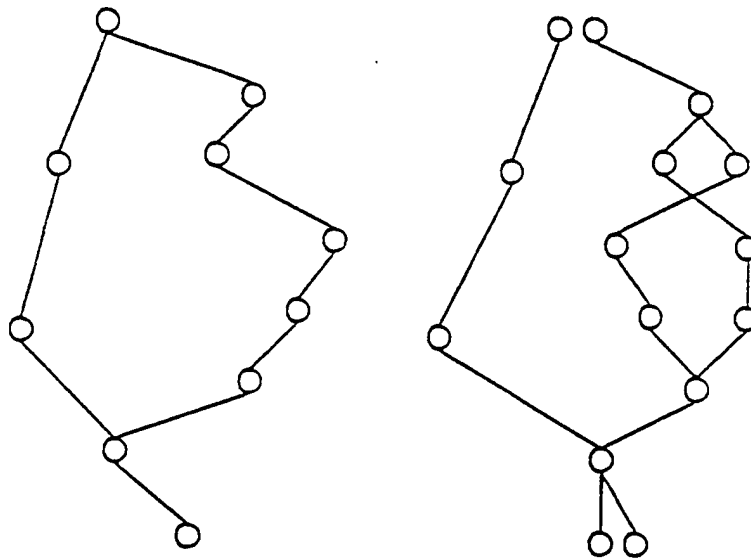Figure 3: An SP graph and its assignment graph.

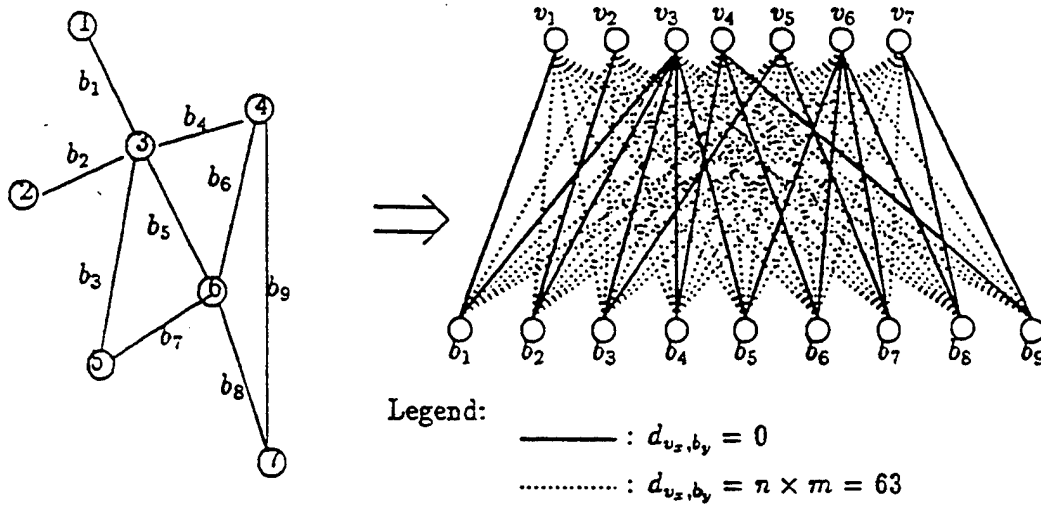Figure 4: An allocation graph and a replication graph of Figure 3.

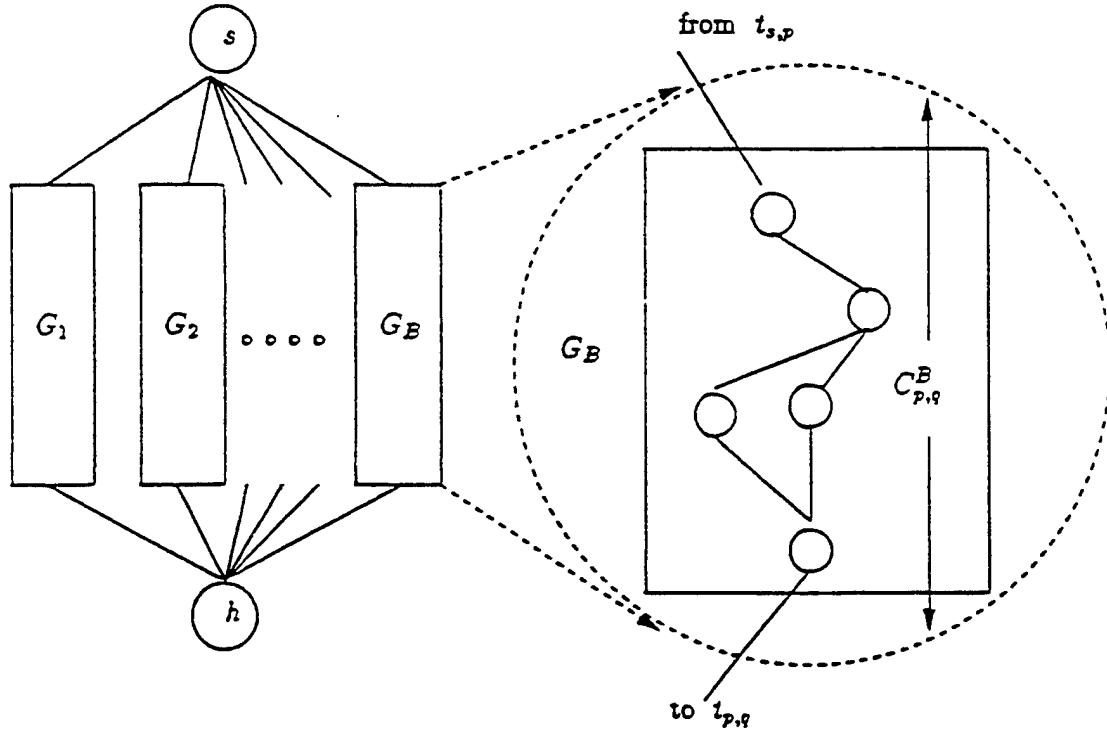Figure 5: An illustration about how to transform a graph to a UCTR instnace

Figure 6: A T_{and} SP graph and the graphical intertrepation of $C_{p,q}^b$.

Legend:

——— : $d_{v_x,b_y} = 0$

·········· : $d_{v_x,b_y} = m \times n = 2 \times 3 = 6$

e table	$p = 1$	$p = 2$	$p = 3$
$e_{s,p} =$	2	2	2
$e_{1,p} =$	0	12	12
$e_{2,p} =$	0	12	12
$e_{3,p} =$	0	12	12
$e_{4,p} =$	0	12	12
$e_{5,p} =$	0	12	12
$e_{6,p} =$	0	12	12
$e_{7,p} =$	0	12	12
$e_{8,p} =$	0	12	12
$e_{9,p} =$	0	12	12
$e_{10,p} =$	0	12	12
$e_{11,p} =$	0	12	12
$e_{12,p} =$	0	12	12
$e_{i,p} =$	0	12	12

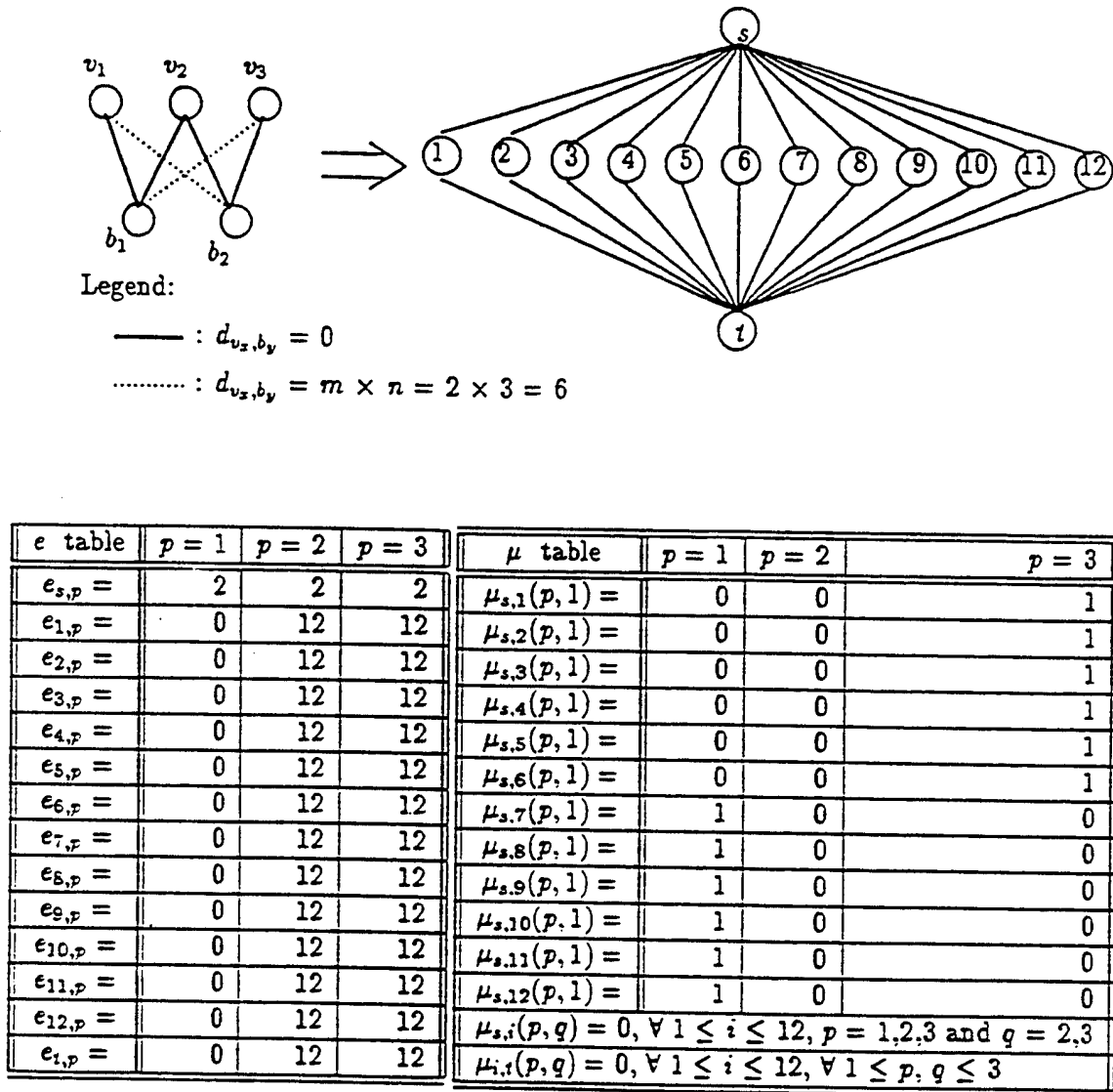| μ table	$p = 1$	$p = 2$	$p = 3$
$\mu_{s,1}(p,1) =$	0	0	1
$\mu_{s,2}(p,1) =$	0	0	1
$\mu_{s,3}(p,1) =$	0	0	1
$\mu_{s,4}(p,1) =$	0	0	1
$\mu_{s,5}(p,1) =$	0	0	1
$\mu_{s,6}(p,1) =$	0	0	1
$\mu_{s,7}(p,1) =$	1	0	0
$\mu_{s,8}(p,1) =$	1	0	0
$\mu_{s,9}(p,1) =$	1	0	0
$\mu_{s,10}(p,1) =$	1	0	0
$\mu_{s,11}(p,1) =$	1	0	0
$\mu_{s,12}(p,1) =$	1	0	0
$\mu_{s,i}(p,q) = 0,\ \forall\ 1 \leq i \leq 12,\ p = 1,2,3$ and $q = 2,3$			
$\mu_{i,i}(p,q) = 0,\ \forall\ 1 \leq i \leq 12,\ \forall\ 1 \leq p, q \leq 3$			

Figure 7: An illustration about how to transform a UCTR instance to a T_{and} SP graph
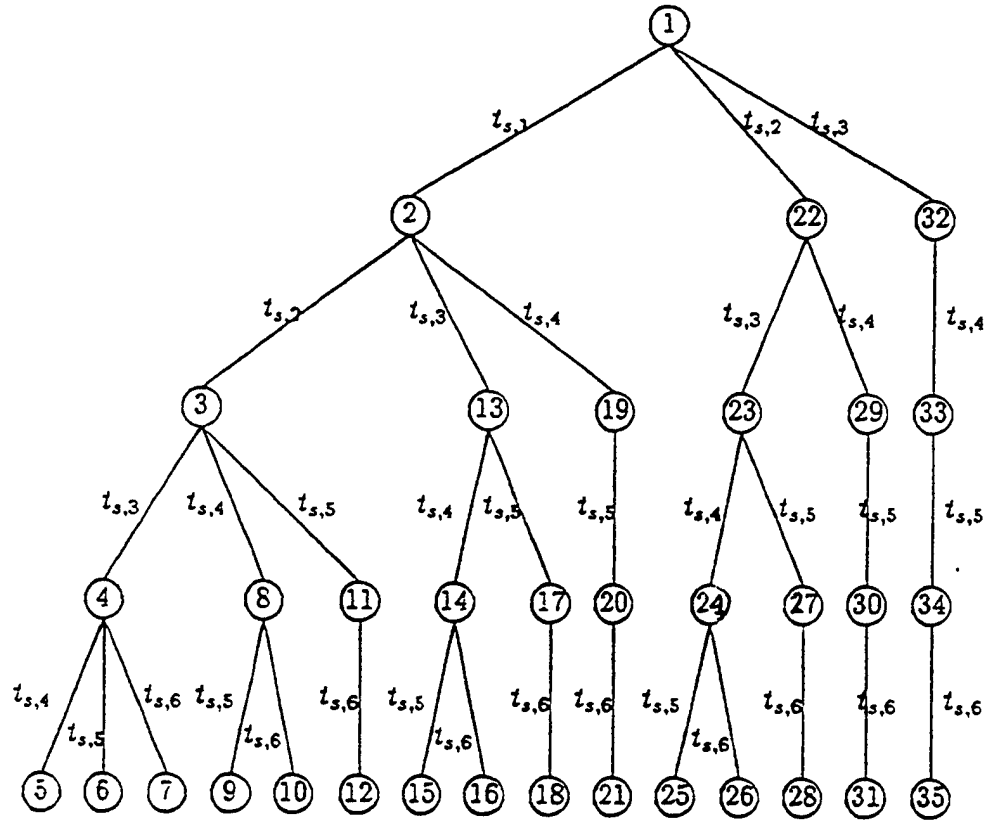
Figure 8: A combination tree for the case where $k = 4$ and $n = 6$

Table 1: Function $BB(k, q, \hat{z})$: branch-and-bound algorithm for solving problem $\mathcal{P}_k^q$

```
01  Initialize the queue to be empty;
02  Insert root node v_0 into the queue;
03  While the queue is not empty do begin
04        Remove the first node u from the queue;
05        Generate all child nodes of u ;
06        For each generated child node v do begin
07              If v is a leaf node (i.e. v is at level k) then
08                    Compute g(v) by setting L to be φ ;
09                    Set ẑ = min ( ẑ, g(v));
10              else begin /* v is an intermediate node */
11                    Compute est(v) by (5) ;
12                    If est(v) < ẑ then
13                          Insert v into the queue according to est(v) ;
14              end;
15        end;
16  end;
17  Return(ẑ).
```

Table 2: Function $OPT(C_{p,q}^b\text{'s}, e_{s,p}\text{'s})$: the optimal solution of MCRP-SP of type T_{and} when $C_{p,q}^b$'s and $e_{s,p}$'s are given

```
01  Sort t_{s,p}'s into a non-decreasing order by values of e_{s,p}'s ;
02  For q = 1 to n do begin
03        Let node v be a leaf node at level 1;
04        Set v to be t_{s,1} and k to be 1;
05        Compute g(v) by setting L to be φ ;
06        Initialize ẑ to be g(v) ;
07        For k = 1 to n do
08              ẑ = BB(k, q, ẑ) ;
09        Set c(q) = ẑ ;
10  end;
11  Output the combination with the minimum value among c(1), c(2), ..., c(n).
```

Figure 9: Pseudo code, graphical demonstration, and dynamic programming table for approximation methods

$Sub(p-1,b) \rightarrow Sub'(p,b)$:

If $e_{s,p} \le \sum_{i=1}^{b}([\min_{x \in Sub(p-1,b)}(C_{x,q}^i)] - C_{p,q}^i)^+$
begin

    $Sub'(p,b) = Sub(p-1,b) \oplus t_{s,p}$

    Reassign&Remove($Sub'(p,b)$)

end

Else $Sub'(p,b) = Sub(p-1,b)$

Legend:

    $(x)^+ = x$, if $x > 0$.

    $(x)^+ = 0$, if $x \le 0$.

$Sub(p,b-1) \rightarrow Sub''(p,b)$:

Let $t_{s,x}$ be the one satisfys $\min_{1 \le i \le p}(C_{i,q}^b)$.

If $t_{s,x} \in Sub(p,b-1)$ then

    $Sub''(p,b) = Sub(p,b-1)$

Else

if $e_{s,x} \le \sum_{i=1}^{b}([\min_{j \in Sub(p,b-1)}(C_{j,q}^i)] - C_{x,q}^i)^+$
begin

    $Sub''(p,b) = Sub(p-1,b) \oplus t_{s,x}$

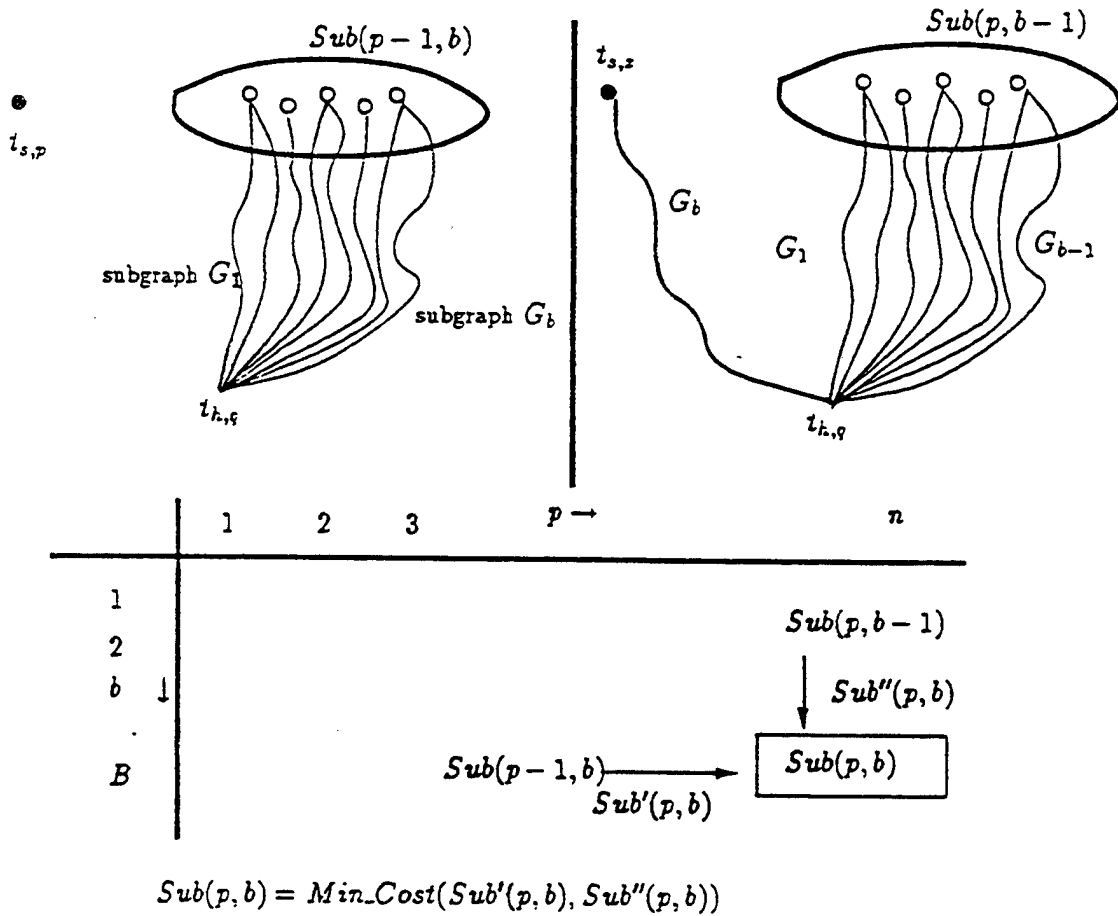   Reassign&Remove($Sub''(p,b)$)

end

Else $Sub''(p,b) = Sub(p,b-1)$

$Sub(p,b) = Min\_Cost(Sub'(p,b), Sub''(p,b))$

Table 3: Computation Results for branch-and-bound approach

n	B	Set I		Set II		Total Number of leaves (2^n)
		EIM[‡]	VLF[‡]	EIM[‡]	VLF[‡]	
4	20	2	6	4	7	16
	24	3	6	3	6	16
	28	4	7	3	6	16
	32	4	7	3	6	16
	36	4	7	4	7	16
	40	3	6	3	6	16
8	20	36	74	16	51	256
	24	40	75	21	62	256
	28	50	86	26	68	256
	32	63	94	37	78	256
	36	73	96	47	84	256
	40	81	97	50	86	256
12	20	186	558	81	340	4,096
	24	231	639	102	398	4,096
	28	349	839	167	543	4,096
	32	451	967	204	617	4,096
	36	454	984	269	720	4,096
	40	636	1,186	301	780	4,096
16	20	758	3,216	203	1,175	65,536
	24	1,065	4,161	329	1,711	65,536
	28	1,335	4,862	546	2,496	65,536
	32	1,884	6,250	726	3,127	65,536
	36	2,322	7,227	839	3,493	65,536
	40	2,880	8,511	1,179	4,510	65,536
20	20	2,026	12,042	389	3,079	1,048,576
	24	3,579	18,866	761	5,280	1,048,576
	28	5,551	27,018	1,227	7,905	1,048,576
	32	6,405	30,521	1,709	10,357	1,048,576
	36	9,517	40,767	2,681	15,032	1,048,576
	40	11,651	48,087	3,086	16,857	1,048,576

[‡]: Each value shown is the average value over 50 runs.

Table 4: Simulation Results for Approximation Method

n	B	SINGLE[‡]	APPROX[‡]	EXHAUST[‡]	single error %	approx error %
4	20	2876	2407	2400	20	0.28
	24	3463	2835	2831	22	0.16
	28	4032	3264	3259	24	0.18
	32	4606	3678	3673	25	0.11
	36	5198	4084	4082	27	0.05
	40	5790	4514	4514	28	0.00
8	20	2794	2282	2250	24	1.46
	24	3356	2672	2636	27	1.38
	28	3931	3060	3028	30	1.05
	32	4540	3443	3413	33	0.88
	36	5127	3831	3800	35	0.80
	40	5683	4215	4192	36	0.55
12	20	2767	2213	2161	28	2.42
	24	3359	2592	2542	32	1.99
	28	3912	2996	2941	33	1.88
	32	4491	3364	3299	36	1.97
	36	5063	3736	3676	38	1.62
	40	5610	4101	4043	39	1.43
16	20	2733	2167	2111	29	2.66
	24	3287	2558	2492	32	2.66
	28	3844	2932	2865	34	2.31
	32	4393	3315	3240	36	2.32
	36	4991	3659	3584	39	2.10
	40	5558	4045	3970	40	1.89

[‡]: Each value shown is the average value over 50 runs.

$$\text{single error\%} = \frac{\text{SINGLE} - \text{EXHAUST}}{\text{EXHAUST}} \times 100\%.$$

$$\text{approx error\%} = \frac{\text{APPROX} - \text{EXHAUST}}{\text{EXHAUST}} \times 100\%.$$

Table 5: Algorithm $FIND(S_x)$: the algorithm for finding the shortest path combinations from the limb which corresponds to the subtree S_x induced by an intermediate node x and all x's descendant nodes in a parsing tree

```
01  Case of the type of intermediate node x:
02    Type T_chain :
03        For b = the first child node of x to the last one do
04            FIND(S_b);  /* Now the limb corresponding to S_b is replaced */
05        Replace the limb corresponding to S_x with a two-layer T_chain limb where
06        the source (sink) layer of the old limb is the source (sink) layer of new 2-layer limb;
07        Put weights on the edges between source and sink layers equal to the shortest path
08        between the corresponding nodes;
09
10    Type T_and : /* Let x = [ T_and, forker s, joiner h ] */
11        Let d be the predecessor of forker s in G (i.e. < d, s > ∈ V);
12        Let B be the number of child nodes of x in the parsing tree;
13        /* I.e. there are B subgraphs connected by s and h */
14        For b = the first child node of x to the B-th child of x do
15            FIND(S_b);  /* Now the limb corresponding to S_b is replaced */
16        For p = 1 to n, q = 1 to n and b = 1 to B do
17            Compute the minimum replication cost C^b_{p,q} from t_{s,p} to t_{h,q} w.r.t. child b :
18        For i = 1 to n do begin
19            For p = 1 to n do  E_{s,p} = μ_{d,s}(i,p) + e_{s,p} ;
20            /* E_{s,p} accounts for initialization by t_{d,i} and execution cost itself. */
21            For q = 1 to n do  μ_{d,h}(i,q) = OPT(C^b_{p,q}'s, E_{s,p}'s) ;
22            /* Create new edges from t_{d,i}'s to t_{h,q}'s */
23        end;
24        Replace the T_and limb with a T_unit limb, where source layer = sink layer = layer h,
25        and there are new edges from layer d to layer h;
26
27    Type T_or : /* Let x = [ T_or, forker s, joiner h ] */
28        Use the same method described above from lines 12 to 17 to compute C^b_{p,q}'s ;
29        Replace the T_or limb with a two-layer T_chain limb, where
30        the source (sink) layer of T_or limb is the source (sink) layer of T_chain limb and
31        μ_{s,h}(p,q) = min_b(C^b_{p,q}), ∀ p and q ;
32  end case;
33  Save the shortest paths between any node in source layer and any node
    in sink layer for future reference.
```

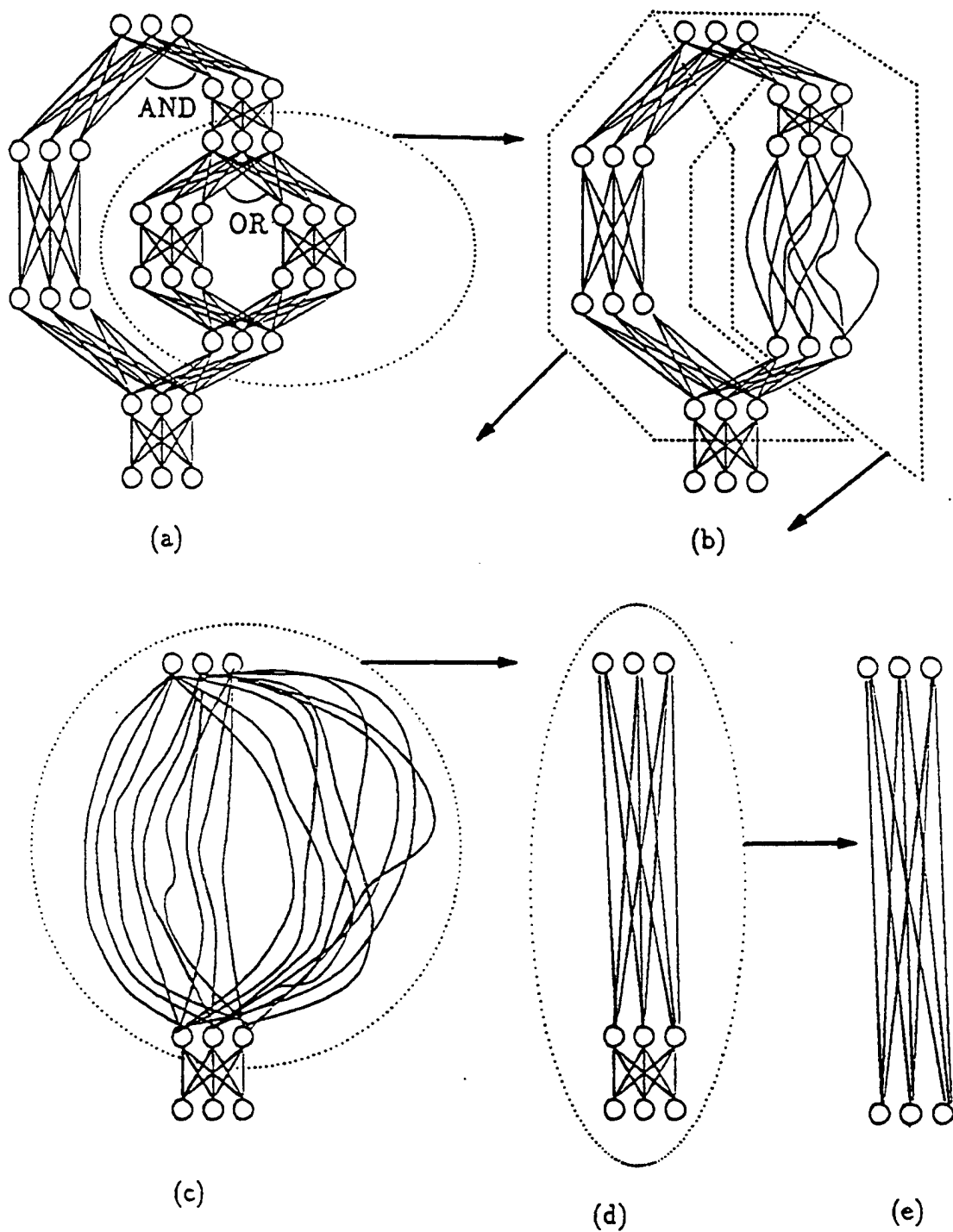Figure 10. A graphical demostration of how to find an optimal solution for MCRP-SP

REPORT DOCUMENTATION PAGE

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE 10/12/94	3. REPORT TYPE AND DATES COVERED Technical

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Optimal Replication of Series-Parallel Graphs for Computation-Intensive Applications Revised Version	N00014-91-C-0195 DASG-60-92-C-0055

6. AUTHOR(S)

Sheng-Tzong Cheng and Ashok K. Agrawala

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Department of Computer Science A. V. Williams Building University of Maryland College Park, MD 20742	Revised Version CS-TR-3020.1 UMIACS-TR-93-4.1

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING / MONITORING AGENCY REPORT NUMBER
Honeywell, Inc. 3600 Technology Drive Minneapolis, MN 55418 Phillips Laboratory Directorate of Contracting 3651 Lowry Avenue SE Kirtland AFB NM 87117-5777	

11. SUPPLEMENTARY NOTES

This version supercedes the previous version.

12a. DISTRIBUTION / AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

We consider the replication problem of series-parallel(SP) task graphs where each task may run on more than one processor. The objective of the problem is to minimize the total cost of task execution and interprocessor communication. We call it, the minimum cost replication problem for SP graphs (MCRP-SP). In this paper, we adopt a new communication model where the purpose of replication is to reduce the total cost. The class of applications we consider is computation-intensive application in which the execution cost of a task is greater than its communication cost. The complexity of MCRP-SP for such applications is proved to be NP-complete. We present a branch-and-bound method to find an optimal solution as well as an approximation approach for suboptimal solution. The numerical results show that such replication may lead to a lower cost than the optimal assignment problem (in which each task is assigned to only one processor) does. The proposed optimal solution has the complexity of $O(n^2 2^n M)$, while the approximation solution has $O(n^4 M^2)$, where n is the number of processors in the system and M is the number of tasks in the graph.

14. SUBJECT TERMS Operating Systems Storage Management, Communications Management	15. NUMBER OF PAGES 35 pages
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclasssified	Unclassified	Unclassified	Unlimited

Designing Temporal Controls *†

Ashok K. Agrawala   Seonho Choi
Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
College Park, MD 20742
{agrawala, seonho}@cs.umd.edu

Leyuan Shi
Department of Industrial Engineering
University of Wisconsin
Madison, WI 53706
leyuan@ie.engr.wisc.edu

Abstract

Traditional control systems have been designed to exercise control at regularly spaced time instants. When a discrete version of the system dynamics is used, a constant sampling interval is assumed and a new control value is calculated and exercised at each time instant. In this paper we formulate a new control scheme, *temporal control*, in which we not only calculate the control value but also decide the time instants when the new values are to be used. Taking a discrete, linear, time-invariant system, and a cost function which reflects a cost for computation of the control values, as an example, we show the feasibility of using this scheme. We formulate the temporal control scheme as a feedback scheme and, through a numerical example, demonstrate the significant reduction in cost through the use of temporal control.

1 Introduction

Control systems have been used for the control of dynamic systems by generating and exercising control signals. Traditional approach for feedback controls has been to define the control signals, $u(t)$, as a function of the current state of the system, $x(t)$. As the state of the system changes continuously the controls change continuously, i.e. they are defined as functions of time, t, such that time is treated as a continuous variable. When computers are used for implementing the control systems, due to the discrete nature of computations, time is treated as a discrete variable obtained by regularly spaced sampling of the time axis at Δ seconds. Many standard control formulations are defined for the discrete version of the system, with system dynamics expressed at discrete time instants. In these formulations the system dynamics and the control are expressed as sequences, $x(k)$ and $u(k)$.

Most of the traditional control systems were designed for dedicated controllers which had only one function, to accept the state values, $x(k)$ and generate the control, $u(k)$. However, when a general purpose computer is used as a controller, it has the capabilities, and may, therefore, be used for other functions. Thus, it may be desirable to take into account the cost of computations and consider control laws which do not compute the new value of the control at every instant. When no control is to be exercised, the computer may be used for other functions. In this paper we formulate such a control law and show how it can be used for control of systems, achieving the same degree of control as traditional control systems while reducing computation costs by changing the control at a few, specific time instants. We term this *temporal control*.

To the best of our knowledge this approach to the design and implementation of controls has not been studied in the past. However, taking computation time delay into consideration for real-time computer control has been studied in several research papers [1, 5, 6, 9, 11, 13]. But, all of these papers concentrated on examining computation time delay effects and compensating them while maintaining the assumption of exercising controls at regularly spaced time instants.

The basic idea of temporal control is to determine not only the values for u but also the time instants at which the values are to be calculated and changed. The control values are assumed to remain constant between changes. By exercising control over the time instants of changes the designer has an additional degree of freedom for optimization. In this paper we present the idea and demonstrate its feasibility through an example using a discrete, linear, and time invariant system. Clearly, the same idea can be extended to continuous time as well as non-linear system.

The paper is organized as follows. In Section 2, we formulate the temporal control problem and introduce computation cost into performance index function. The solution approach for temporal control scheme is discussed in *Section 3*. In Section 4, implementation issues are addressed. We

provide an example of controlling rigid body satellite in Section 5 . In this example, an optimal temporal controller is designed. Results show that the temporal control approach performs better than the traditional sampled data control approach with the same number of control exercises. Section 6 deals with the application of temporal controls to the design of real-time control systems. Finally, Section 7, we present our conclusions.

2   Problem Formulation

In temporal control, the number of control changes and their exercising time instants within the controlling interval $[0, T_f]$ is decided to minimize a cost function. To formulate the temporal control problem for a discrete, linear time-invariant system, we first discretize the time interval $[0, T_f]$ into M subintervals of length $\Delta = T_f/M$. Let $D_M = \{0, \Delta, 2\Delta, ..., (M-1)\Delta\}$ which denote M time instants which are regularly spaced. Here, control exercising time instants are restricted within D_M for the purpose of simplicity. The linear time-invariant controlled process is described by the difference equation:

$$
\begin{aligned}
x(k+1) &= Ax(k) + Bu(k) \\
y(k) &= Cx(k)
\end{aligned}
\tag{1}
$$

where k is the time index. One unit of time represents the subinterval Δ, whereas $x \in \mathcal{R}^n$ and $u \in \mathcal{R}^l$ are the state and input vectors respectively.

It is well known that there exists an optimal control law [4]

$$
u^o(i) = f[x(i)] \quad i = 0, 1, ..., M - 1
\tag{2}
$$

that minimizes the quadratic performance index function (Cost)

$$
J_M = \sum_{k=0}^{M-1} [x^T(k)Qx(k) + u^T(k)Ru(k)] + x^T(M)Qx(M)
\tag{3}
$$

where $Q \in \mathcal{R}^{n \times n}$ is positive semi-definite and $R \in \mathcal{R}^{l \times l}$ is positive definite.

As we can see, traditional controller exercises control at every time instant in D_M. However, in temporal control, we are no longer constrained to exercise control at every time instant in D_M. Therefore, we want to find an optimal control law, δ and g for $i = 0, 1, ..., M - 1$:

$$
\begin{aligned}
u^o(i) &= u^o(i-1) \quad if \ \delta(i) = 0 \\
u^o(i) &= g[x(i)] \quad if \ \delta(i) = 1
\end{aligned}
\tag{4}
$$

41

that minimizes a new performance index function

$$
\begin{aligned}
J'_M &= \sum_{k=0}^{M-1} [x^T(k)Qx(k) + u^T(k)Ru(k)] + x^T(M)Qx(M) + \sum_{k=0}^{M-1} \delta(k)\mu \qquad (5) \\
&= J_M + C_M
\end{aligned}
$$

Here, μ is the computation cost of getting a new control value at a time instant, and $C_M = \sum_{k=0}^{M-1} \delta(k)\mu$ denotes the total computation cost. Note that $\nu = \sum_{k=0}^{M-1} \delta(k)$ is the number of control changes. Also, let $D_\nu = \{t_0, t_1, t_2, \ldots, t_{\nu-1}\}$ consist of control changing time instants where $t_0 = 0$, $t_1 = n_1\Delta$, ..., $t_{\nu-1} = n_{\nu-1}\Delta$. That is, $n_0, n_1, n_2, \ldots, n_{\nu-1}$ are the indices for control changing time instants and $\delta(n_i) = 1$ for $i = 0, 1, 2, \ldots \nu - 1$.

With this new setting we need to choose ν, D_ν, and control input values to find an optimal controller which minimizes J'_M. This new cost function is different from J_M in two aspects. First, the concept of computational cost is introduced in J'_M as C_M term to regulate the number of control changes chosen. If we do not take this computation cost into consideration ν is likely to become M. If computation cost is high (i.e., μ has a large value) then ν is likely to be small in order to minimize the total cost function. Second, in temporal control, not only do we seek optimal control law $u(x(t))$, but also the control exercising time instants and the number of control changes. In the next section, we present in detail specific techniques for finding an optimal temporal control law.

3 Temporal Control

We develop a three-step procedure for finding an optimal temporal controller.

Step 1. Find an optimal control law given ν and D_ν
Step 2. Find best D_ν given ν
Step 3. Find best ν

First, in the following two subsections(3.1 and 3.2) we derive a temporal control law which minimizes the cost function J'_M when D_ν is given, i.e., both time instants and number of controls are fixed. Since ν and D_ν are fixed we can use J_M defined in (5) as a cost function instead of J'_M. Secondly, assume that ν is fixed but D_ν can vary. Then we present an algorithm in section 3.3 to find a D_ν^o such that J_M (and J'_M) is minimized. Finally, we will vary ν from 1 to ν_{max} to search an optimal D_ν^o at which temporal control should be exercised. Section 3.4 presents this iteration procedure. Section 3.5 explains how to incorporate terminal state constraints into the above procedure of getting an optimal temporal control law. And a complete algorithm of the

above procedure is described in Section 3.6. Finally, in Section 3.7 we explain how to get optimal temporal controllers over an initial state space.

3.1 Closed-loop Temporal Control with D_ν Given

Assume that ν and D_ν are given. Then a new control input calculated at t_i will be applied to the actuator for the next time interval from t_i to t_{i+1}. Our objective here is to determine the optimal control law

$$u^o(n_i) = g[x(n_i)] \quad i = 0, 1, ..., \nu - 1 \tag{6}$$

that minimizes the quadratic performance index function (Cost) J_M which is defined in (5).
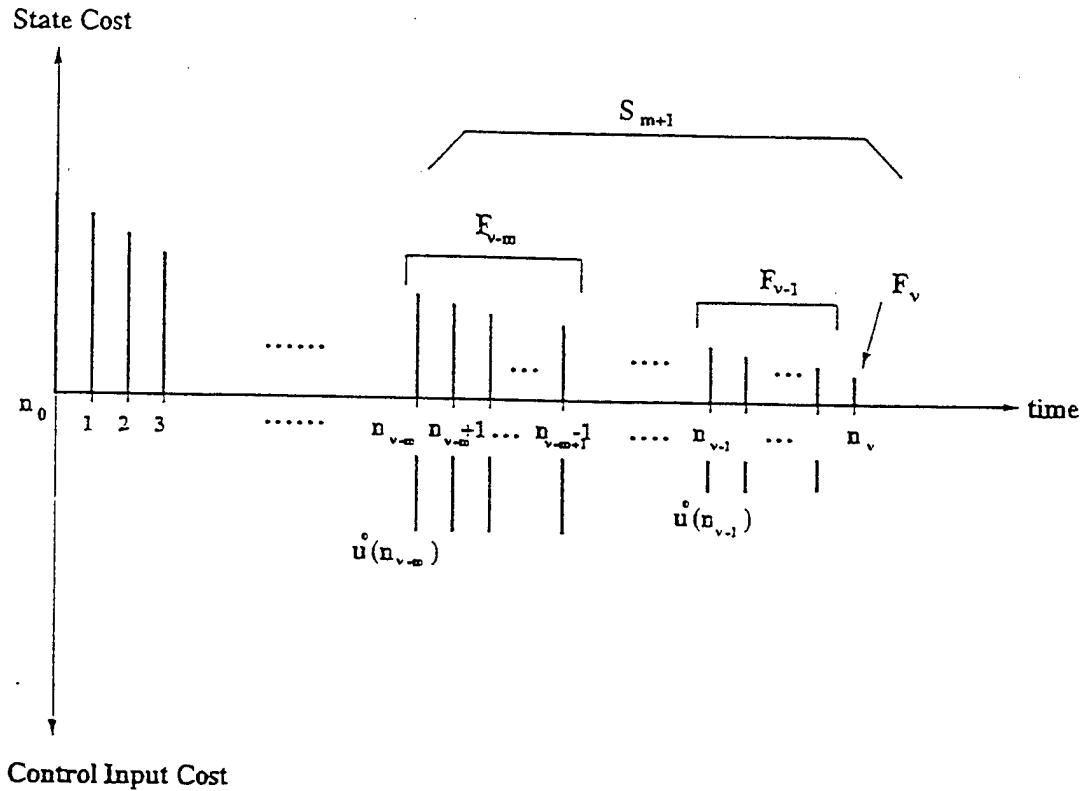
Figure 1: Decomposition of J_M into F_i.

The principle of optimality, developed by Richard Bellman[2, 3] is the approach used here. That is, if a closed loop control $u^o(n_i) = g[x(n_i)]$ is optimal over the interval $t_0 \leq t \leq t_\nu$, then it is also optimal over any sub-interval $t_m \leq t \leq t_\nu$, where $0 \leq m \leq \nu$. As it can be seen from Figure 1, the

43

total cost J_M can be decomposed into F_is for $0 \leq i \leq \nu$ where

$$
\begin{aligned}
F_i &= x^T(n_i)Qx(n_i) + x^T(n_i + 1)Qx(n_i + 1) \\
&\quad + x^T(n_i + 2)Qx(n_i + 2) + ... + x^T(n_{i+1} - 1)Qx(n_{i+1} - 1) \\
&\quad + (n_{i+1} - n_i)u^T(n_i)Ru(n_i)
\end{aligned}
\tag{7}
$$

That is, from (1),

$$
\begin{aligned}
F_i &= x^T(n_i)Qx(n_i) + (Ax(n_i) + Bu(n_i))^T Q(Ax(n_i) + Bu(n_i)) \\
&\quad + (A^2 x(n_i) + ABu(n_i) + Bu(n_i))^T Q(A^2 x(n_i) + ABu(n_i) + Bu(n_i)) \\
&\quad + ... + (A^{n_{i+1} - n_i - 1} x(n_i) + A^{n_{i+1} - n_i - 2}Bu(n_i) + ... + ABu(n_i) + Bu(n_i))^T Q \\
&\quad\quad (A^{n_{i+1} - n_i - 1} x(n_i) + A^{n_{i+1} - n_i - 2}Bu(n_i) + ... + ABu(n_i) + Bu(n_i)) \\
&\quad + (n_{i+1} - n_i)u^T(n_i)Ru(n_i)
\end{aligned}
\tag{8}
$$

This can be rewritten as

$$
F_i = x^T(n_i)Qx(n_i) + \sum_{j=1}^{n_{i+1} - n_i - 1} [A_j x(n_i) + B_j u(n_i)]^T Q[A_j x(n_i) + B_j u(n_i)] \tag{9}
$$
$$
+ (n_{i+1} - n_i)u^T(n_i)Ru(n_i)
$$

where $A_j = A^j$ and $B_j = \sum_{k=0}^{j-1} A^k B$.

Then J_M can be expressed as

$$
J_M = F_0 + F_1 + F_2 + ... + F_\nu. \tag{10}
$$

Let S_m be the cost from $i = \nu - m + 1$ to $i = \nu$:

$$
S_m = F_{\nu - m + 1} + F_{\nu - m + 2} + ... + F_{\nu - 1} + F_\nu; \quad 1 \leq m \leq \nu + 1. \tag{11}
$$

These cost terms are well illustrated in the above Figure 1.

Therefore, by applying the principle of optimality, we can first minimize $S_1 = F_\nu$, then choose $F_{\nu-1}$ to minimize $S_2 = F_{\nu-1} + F_\nu = S_1^o + F_{\nu-1}$ where S_1^o is the optimal cost occurred at t_ν. We can continue choosing $F_{\nu-2}$ to minimize $S_3 = F_{\nu-2} + F_{\nu-1} + F_\nu = F_{\nu-2} + S_2^o$ and so on until $S_{\nu+1} = J_M$ is minimized. Note that $S_1 = F_\nu = x^T(n_\nu)Qx(n_\nu)$ is determined only from $x(n_\nu)$ which is independent of any other control inputs.

44

3.2 Inductive Construction of an Optimal Control Law with D_ν Given

We inductively derive an optimal controller which changes its control at ν time instants $t_0, t_1, \ldots, t_{\nu-1}$. As we showed in the previous section, the inductive procedure goes backwards in time from S_1^o to $S_{\nu+1}^o$. Since $S_1 = F_\nu = x^T(n_\nu)Qx(n_\nu) + u^T(n_\nu)Ru(n_\nu)$ and $x(n_\nu)$ is independent of $u(n_\nu)$, we can let $u^o(n_\nu) = u^o(M) = 0$ and $S_1^o = x^T(n_\nu)Qx(n_\nu)$ where Q is symmetric and positive semi-definite.

Induction Basis: $S_1^o = x^T(n_\nu)Qx(n_\nu)$ where Q is symmetric.

Inductive Assumption: Suppose that
$$S_m^o = x^T(n_{\nu-m+1})P(\nu - m + 1)x(n_{\nu-m+1})$$
holds for some m where $1 \leq m \leq \nu$ and $P(\nu - m + 1)$ is symmetric.

We can write S_m^o as

$$
\begin{aligned}
S_m^o \;=\; & [A_{(n_{\nu-m+1}-n_{\nu-m})}x(n_{\nu-m}) + B_{(n_{\nu-m+1}-n_{\nu-m})}u(n_{\nu-m})]^T P(\nu - m + 1) \\
& [A_{(n_{\nu-m+1}-n_{\nu-m})}x(n_{\nu-m}) + B_{(n_{\nu-m+1}-n_{\nu-m})}u(n_{\nu-m})]
\end{aligned}
\tag{12}
$$

From the definition of S_m and (9),

$$
\begin{aligned}
S_{m+1} \;=\; & S_m^o + F_{\nu-m} \\
\;=\; & S_m^o + x^T(n_{\nu-m})Qx(n_{\nu-m}) \\
& + \sum_{j=1}^{n_{\nu-m+1}-n_{\nu-m}-1} [A_j x(n_{\nu-m}) + B_j u(n_{\nu-m})]^T Q[A_j x(n_{\nu-m}) + B_j u(n_{\nu-m})] \\
& + (n_{\nu-m+1} - n_{\nu-m})u^T(n_{\nu-m})Ru(n_{\nu-m})
\end{aligned}
\tag{13}
$$

And the above equation becomes

$$
\begin{aligned}
S_{m+1} \;=\; & [A_{n_{\nu-m+1}-n_{\nu-m}}x(n_{\nu-m}) + B_{n_{\nu-m+1}-n_{\nu-m}}u(n_{\nu-m})]^T P(\nu - m + 1) \\
& [A_{n_{\nu-m+1}-n_{\nu-m}}x(n_{\nu-m}) + B_{n_{\nu-m+1}-n_{\nu-m}}u(n_{\nu-m})] \\
& + x^T(n_{\nu-m})Qx(n_{\nu-m}) \\
& + \sum_{j=1}^{n_{\nu-m+1}-n_{\nu-m}-1} [A_j x(n_{\nu-m}) + B_j u(n_{\nu-m})]^T Q[A_j x(n_{\nu-m}) + B_j u(n_{\nu-m})] \\
& + (n_{\nu-m+1} - n_{\nu-m})u^T(n_{\nu-m})Ru(n_{\nu-m})
\end{aligned}
\tag{14}
$$

If we differentiate S_{m+1} with respect to $u(n_{\nu-m})$, then

$$
\begin{aligned}
\frac{\partial S_{m+1}}{\partial u(n_{\nu-m})} = & \; B^T_{n_{\nu-m+1}-n_{\nu-m}} P(\nu-m+1) A_{n_{\nu-m+1}-n_{\nu-m}} x(n_{\nu-m}) \\
& + (A^T_{n_{\nu-m+1}-n_{\nu-m}} P(\nu-m+1) B_{n_{\nu-m+1}-n_{\nu-m}})^T x(n_{\nu-m}) \\
& + 2 B^T_{n_{\nu-m+1}-n_{\nu-m}} P(\nu-m+1) B_{n_{\nu-m+1}-n_{\nu-m}} u(n_{\nu-m}) \\
& + \sum_{j=1}^{n_{\nu-m+1}-n_{\nu-m}-1} [2 B_j^T Q A_j x(n_{\nu-m}) + 2 B_j^T Q B_j u(n_{\nu-m})] \\
& + 2(n_{\nu-m+1}-n_{\nu-m}) R u(n_{\nu-m}) \\
= & \; 2\{ B^T_{n_{\nu-m+1}-n_{\nu-m}} P(\nu-m+1) A_{n_{\nu-m+1}-n_{\nu-m}} \\
& + \sum_{j=1}^{n_{\nu-m+1}-n_{\nu-m}-1} B_j^T Q A_j \} x(n_{\nu-m}) \\
& + 2\{ B^T_{n_{\nu-m+1}-n_{\nu-m}} P(\nu-m+1) B_{n_{\nu-m+1}-n_{\nu-m}} \\
& + \sum_{j=1}^{n_{\nu-m+1}-n_{\nu-m}-1} B_j^T Q B_j + (n_{\nu-m+1}-n_{\nu-m}) R \} u(n_{\nu-m})
\end{aligned}
$$

(15)

(16)

Note that $P(\nu-m+1)$ is symmetric and the following three rules are applied to differentiate S_{m+1} above.

$$
\begin{aligned}
\frac{\partial}{\partial x}(x^T Q x) &= 2 Q x \\
\frac{\partial}{\partial x}(x^T Q y) &= Q y \\
\frac{\partial}{\partial y}(x^T Q y) &= Q^T x
\end{aligned}
$$

Let $\frac{\partial S_{m+1}}{\partial u(n_{\nu-m})} = 0$, from Lemma 1 and Lemma 2 given later we can obtain $u^o(n_{\nu-m})$ which minimizes S_{m+1} and thus obtain S^o_{m+1}.

$$
\begin{aligned}
u^o(n_{\nu-m}) = & \; -\{ B^T_{n_{\nu-m+1}-n_{\nu-m}} P(\nu-m+1) B_{n_{\nu-m+1}-n_{\nu-m}} \\
& + \sum_{j=1}^{n_{\nu-m+1}-n_{\nu-m}-1} B_j^T Q B_j + (n_{\nu-m+1}-n_{\nu-m}) R \}^{-1} \\
& \{ B^T_{n_{\nu-m+1}-n_{\nu-m}} P(\nu-m+1) A_{n_{\nu-m+1}-n_{\nu-m}} + \sum_{j=1}^{n_{\nu-m+1}-n_{\nu-m}-1} B_j^T Q A_j \} x(n_{\nu-m}) \\
= & \; -K(\nu-m) x(n_{\nu-m})
\end{aligned}
$$

(17)

where $K(\nu-m)$ is defined in (17).

Therefore, we can write

$$A_{n_{\nu-m+1}-n_{\nu-m}}x(n_{\nu-m}) \;+\; B_{n_{\nu-m+1}-n_{\nu-m}}u^o(n_{\nu-m}) = \tag{18}$$

$$[A_{n_{\nu-m+1}-n_{\nu-m}} - B_{n_{\nu-m+1}-n_{\nu-m}}K(\nu-m)]x(n_{\nu-m})$$

If we use (17) and (18), we have

$$
\begin{aligned}
S^o_{m+1} \;=\;& \{[A_{n_{\nu-m+1}-n_{\nu-m}} - B_{n_{\nu-m+1}-n_{\nu-m}}K(\nu-m)]x(n_{\nu-m})\}^T P(\nu-m+1) \tag{19}\\
& \{[A_{n_{\nu-m+1}-n_{\nu-m}} - B_{n_{\nu-m+1}-n_{\nu-m}}K(\nu-m)]x(n_{\nu-m})\}\\
& +\; x^T(n_{\nu-m})Qx(n_{\nu-m})\\
& +\; \sum_{j=1}^{n_{\nu-m+1}-n_{\nu-m}-1}\{[A_j - B_jK(\nu-m)]x(n_{\nu-m})\}^T Q\{[A_j - B_jK(\nu-m)]x(n_{\nu-m})\}\\
& +\; (n_{\nu-m+1} - n_{\nu-m})[K(\nu-m)x(n_{\nu-m})]^T R[K(\nu-m)x(n_{\nu-m})]
\end{aligned}
$$

This equation can be rewritten as

$$
\begin{aligned}
S^o_{m+1} \;=\;& x^T(n_{\nu-m})\{[A_{n_{\nu-m+1}-n_{\nu-m}} - B_{n_{\nu-m+1}-n_{\nu-m}}K(\nu-m)]^T P(\nu-m+1) \tag{20}\\
& [A_{n_{\nu-m+1}-n_{\nu-m}} - B_{n_{\nu-m+1}-n_{\nu-m}}K(\nu-m)]\\
& +\; Q\\
& +\; \sum_{j=1}^{n_{\nu-m+1}-n_{\nu-m}-1}[A_j - B_jK(\nu-m)]^T Q[A_j - B_jK(\nu-m)]\\
& +\; (n_{\nu-m+1} - n_{\nu-m})K^T(n_{\nu-m})RK(\nu-m)\}x(n_{\nu-m}).\\
\;=\;& x^T(n_{\nu-m})P(\nu-m)x(n_{\nu-m})
\end{aligned}
$$

where $P(\nu-m)$ is obtained from $K(\nu-m)$ and $P(\nu-m+1)$ as in (20). Also note that knowing $P(\nu-m+1)$ is enough to compute $K(\nu-m)$ because other terms of (17) are known a priori.

Therefore, we find a symmetric matrix $P(\nu-m)$ satisfying $S^o_{m+1} = x^T(n_{\nu-m})P(\nu-m)x(n_{\nu-m})$. From (17) and (20), we have the following recursive equations for obtaining $P(\nu-m)$ from $P(\nu-m+1)$ where $m = 1, 2, ..., \nu$.

$$
\begin{aligned}
K(\nu-m) \;=\;& \{B^T_{n_{\nu-m+1}-n_{\nu-m}}P(\nu-m+1)B_{n_{\nu-m+1}-n_{\nu-m}} \tag{21}\\
& +\; \sum_{j=1}^{n_{\nu-m+1}-n_{\nu-m}-1} B^T_j QB_j + (n_{\nu-m+1} - n_{\nu-m})R\}^{-1}\\
& \{B^T_{n_{\nu-m+1}-n_{\nu-m}}P(\nu-m+1)A_{n_{\nu-m+1}-n_{\nu-m}} + \sum_{j=1}^{n_{\nu-m+1}-n_{\nu-m}-1} B^T_j QA_j\}
\end{aligned}
$$

47

$$P(\nu - m) = [A_{n_{\nu-m+1}-n_{\nu-m}} - B_{n_{\nu-m+1}-n_{\nu-m}}K(\nu - m)]^T P(\nu - m + 1) \tag{22}$$
$$[A_{n_{\nu-m+1}-n_{\nu-m}} - B_{n_{\nu-m+1}-n_{\nu-m}}K(\nu - m)]$$
$$+ \ Q$$
$$\div \ \sum_{j=1}^{n_{\nu-m+1}-n_{\nu-m}-1} [A_j - B_j K(\nu - m)]^T Q[A_j - B_j K(\nu - m)]$$
$$+ \ (n_{\nu-m+1} - n_{\nu-m})K^T(\nu - m)RK(\nu - m)$$

Also, we know that at each time instant $n_{\nu-m}\Delta$

$$u^o(n_{\nu-m}) = -K(\nu - m)x(n_{\nu-m}) \tag{23}$$

Hence, with $P(\nu) = Q$, we can obtain $K(i)$ and $P(i)$ for $i = \nu - 1, \nu - 2, ..., 0$ recursively using (21) and (22). At each time instant $n_i\Delta$, $i = 0, 1, 2, ..., \nu - 1$ the new control input value will be obtained using (23) by multiplying $K(i)$ by $x(n_i)$ where $x(n_i)$ is the estimate of the system state at $n_i\Delta$. Also, note that the optimal control cost is $J_M^o = S_{\nu+1}^o = x^T(0)P(0)x(0)$ where $P(0)$ is found from the above procedure.

To prove the optimality of this control law we need the following lemmas.

Lemma 1 *If Q is positive semi-definite and R is positive definite, then $P(i)$, $i = \nu, \nu-1, \nu-2, ..., 0$, matrices are positive semi-definite. Hence, $P(i)s$ are symmetric from the definition of a positive semi-definite matrix.*

Proof Since $P(\nu) = Q$, from assumption $P(\nu)$ is positive semi-definite. Assume that for $k = i + 1$, $P(k)$ is positive semi-definite. We use induction to prove that $P(i)$ is semi-definite. Note that Q is positive semi-definite and R is positive definite. From (22) we have

$$P(i) = [A_{n_{i+1}-n_i} - B_{n_{i+1}-n_i}K(i)]^T P(i + 1) \tag{24}$$
$$[A_{n_{i+1}-n_i} - B_{n_{i+1}-n_i}K(i)]$$
$$+ \ Q$$
$$+ \ \sum_{j=1}^{n_{i+1}-n_i-1} [A_j - B_j K(i)]^T Q[A_j - B_j K(i)]$$
$$+ \ (n_{i+1} - n_i)K^T(i)RK(i)$$

48

Since $P(i+1)$ and Q are positive semi-definite, R is positive definite, and $(n_{i+1} - n_i) > 0$, it is easy to verify that for $\forall y \in R^m$: $y^T P(i) y \geq 0$. This means that $P(i)$ is positive semi-definite. This inductive procedure proves the lemma.

Lemma 2 *Given D_ν, the inverse matrix in (21) always exists.*

Proof   Let $V = B^T_{n_{\nu-m+1} - n_{\nu-m}} P(\nu - m + 1) B_{n_{\nu-m+1} - n_{\nu-m}} + \sum_{j=1}^{n_{\nu-m+1} - n_{\nu-m} - 1} B_j^T Q B_j + (n_{\nu-m+1} - n_{\nu-m}) R$. From Lemma 1, $P(\nu - m + 1)$ is positive semi-definite. Therefore, $\forall y \in R^m$: $y^T V y > 0$ because Q is positive semi-definite, R is positive definite and $n_{\nu-m+1} - n_{\nu-m} > 0$. This implies that V is positive definite. Hence the inverse matrix exists.

Theorem 1 *Given D_ν, $K(i)$ $(i = 0, 1, 2, ..., \nu-1)$ obtained from the above procedure are the optimal feedback gains which minimize the cost function J_M (and J'_M) on $[0, M\Delta]$.*

Proof   Note that given D_ν, J_M is a convex function of $u(n_i), i = 0, 1, ..., \nu - 1$. Thus the above feedback control law is optimal.

Lemma 3 *If $p \leq q$ and $D_p \subseteq D_q$, then $J^o_{M_p} \geq J^o_{M_q}$ where $J^o_{M_p}$ and $J^o_{M_q}$ are the optimal costs of controls which change controls at time instants in D_p and D_q respectively.*

Proof   Suppose that $J^o_{M_p} < J^o_{M_q}$; then, in controlling the system with D_q, if we do not change controls at time instants in $D_q - D_p$ and change controls at time instants in D_p to the same control inputs that were exercised to get $J^o_{M_p}$ with D_p; we obtain $\hat{J}_{M_q}$ which is equal to $J^o_{M_p}$. This contradicts the fact that $J^o_{M_q}$ is the minimum cost obtainable with D_q since we have found $\hat{J}_{M_q}$ which is equal to $J^o_{M_p}$ and therefore less than $J^o_{M_q}$. Hence, $J^o_{M_p} \geq J^o_{M_q}$.

This lemma implies that if we do not take computation cost, μ, into consideration, then the more control exercising points, the better the controller is (less cost). With the computation cost being included in the cost function, the statement above is no longer true. Therefore we need to search for an optimal D_ν which minimizes the cost function J'_M. The following sections provide a detailed discussion on searching for such an optimal solution. Note that if we let $D_\nu = D_M$ then the optimal temporal control law is the same as the traditional linear feedback optimal control law.

3.3 Optimal Temporal Control Law over D_ν Space with ν Given

When the number of control changing points, ν, and an initial system state $x(0)$ are given, we search over a set of possible D_νs and $u(D_\nu)$s such that the cost function J_M is minimized. This can be done by varying $\nu - 1$ control changing time instants, t_i, $i = 1, 2, ..., \nu - 1$ (since $t_0 = 0$) over the discrete set, $D_M = \{0, \Delta, 2\Delta, ..., (M-1)\Delta\}$ and applying the technique developed in the previous section for each given D_ν. Let us denote such a D_ν which minimizes J_M as D_ν^o. Note that when ν is given, minimizing J_M is equivalent to minimizing J_M'. Since both D_ν and $u(D_\nu)$ are control variates, to be able to find a global optimal solution, either an exhaustive search or some global search methods like *Genetic Algorithm* or *Simulated Annealing* should be considered. Later we present a numerical example in which an exhaustive search with *Steepest Descent Search* method is used. Searching for a globally optimal solution for a temporal controller calls for further research.

3.4 Optimal Temporal Control Law

Assume that a maximum number of control changing points, ν_{max}, is given. By varying ν from 1 to ν_{max} we can find D_ν^o to obtain a globally optimal temporal controller which minimizes J_M'. This can be done by first searching for D_ν^o for each given ν and then comparing the cost function $J_M' = J_M + \nu\mu$ at each D_ν^o, $\nu = 1, 2, ..., \nu_{max}$. That is, let $J_{M_\nu}'^o = x^T(0)P(0)x(0) + \nu\mu$ where $P(0)$ is calculated at D_ν^o as in the previous section. Then we can obtain a global minimum cost $J_M'^o = \min_{1 \le \nu \le \nu_{max}}\{J_{M_\nu}'^o\}$ and an optimal number of control changes, ν^o, at which $J_{M_{\nu^o}}'^o = J_M'^o$.

3.5 Terminal State Constraints

The terminal state constraints may be used to check if the optimal temporal controller with D_ν^o can drive the system state to a permissible final state within a given time. Let X_f be a set of allowed terminal states, if $x(n_\nu) \in X_f$, then the control law is said to be *stable* in terms of the terminal state constraints and *not stable* if $x(n_\nu) \notin X_f$. If the globally optimal temporal controller obtained from the above procedure is not stable, ν^* should be increased until a stable one is found. One way of specifying terminal state constraints for regulators might be $\mid x(M)_i \mid \le \epsilon_i$ where $x(M)_i$ is the ith element of $x(M)$ state vector.

3.6 Algorithm to Derive an Optimal Temporal Controller

To summarize the above discussion, we provide in Figure 2 a complete algorithm to search for a globally optimal temporal controller under the assumption that the initial state $x(0)$ is given.

In the algorithm, a neighbor of $D_\nu = \{n_0\Delta, n_1\Delta, n_2\Delta, \ldots, n_{\nu-1}\Delta\}$ is defined to be any member of a set $N(D_\nu) = \{\{n_0'\Delta, n_1'\Delta, \ldots, n_{\nu-1}'\Delta\} \mid |n_i' - n_i| \le 1, i = 1, 2, \ldots, \nu-1\}$.

3.7 Optimal Temporal Controllers over an Initial State Space

Note that D_ν^o might become different if a new initial system state $\hat{x}(0)$ is used instead of $x(0)$ when the state vector is in $R^{m \times 1}$ where $m \ge 2$. This is because the cost function $J_M = x^T(0)P(0)x(0)$ depends on $x(0)$ as well as $P(0)$. Thus, D_ν^o is dependent on the initial state $x(0)$. However, when $m = 1$ it can be shown that D_ν^o is independent of any initial state. To see this let $x(0) = k\hat{x}(0) \in \mathcal{R}^1$ and $P(0)$ and $\hat{P}(0)$ be the optimal matrices with initial states $x(0)$ and $\hat{x}(0)$, respectively. i.e.,

$$J_M(x(0)) = x(0)P(0)x(0)$$

$$J_M(\hat{x}(0)) = \hat{x}(0)\hat{P}(0)\hat{x}(0)$$

From the optimality of $\hat{P}(0)$ with respect to $\hat{x}(0)$,

$$\hat{x}^T(0)P(0)\hat{x}(0) \ge \hat{x}^T(0)\hat{P}(0)\hat{x}(0) \tag{25}$$

Multiplying the above inequality by k^2 we have

$$
\begin{aligned}
k^2\hat{x}^T(0)P(0)\hat{x}(0) &= x^T(0)P(0)x(0) \\
&\ge k^2\hat{x}^T(0)\hat{P}(0)\hat{x}(0) \\
&= x^T(0)\hat{P}(0)x(0)
\end{aligned} \tag{26}
$$

On the other hand, due to the optimality of $P(0)$ we have

$$x^T(0)\hat{P}(0)x(0) \ge x^T(0)P(0)x(0) \tag{27}$$

Therefore, $\hat{P}(0) = P(0)$. This implies the optimality of $\hat{P}(0)$ and $\hat{D}_\nu^o$ for any initial state $x(0) \in \mathcal{R}^1$.

Generally speaking, the above result will not hold for $m \ge 2$ cases. However, using the same argument discussed above we can prove that for any initial state $x(0) = k\hat{x}(0)$, $x(0)$ and $\hat{x}(0)$ will have the same D_ν^o as well as the same $P(0)$.

```
ν° = 1
J'°_M = ∞ ·
for ν = 1 to ν_max {
    /* Several different search starting points */
    for i = 1 to NumInitPts_ν {
        D_ν = D_ν^{init,i}
        /* Iterate until a local minimum is found – Steepest Descent Search */
        while (MinimumFound != True) {
            Find optimal costs for neighboring points of D_ν using theorem 1
            if ( J'_M has a Local Minimum at D_ν)
                then {
                    MinimumFound = True
                    J'^i_{M_ν} = Cost(J'_M) at D_ν }
            else
                    D_ν = a neighbor of D_ν with the smallest J'_M
        }
    }
    J'°_{M_ν} = min_{1≤i≤NumInitPts_ν} {J'^i_{M_ν}}
    if ( J'°_{M_ν} < J'°_M )
        then {
            ν° = ν
            J'°_M = J'°_{M_ν} }
}
```

Figure 2: Complete algorithm to find an optimal temporal controller.

4 Implementation

To implement temporal control, we need to calculate and store $K(i)$ matrices in (22) and use them when controlling the system utilizing (23). Note that in traditional optimal linear control a similar matrix is obtained and used at every time instant in D_M to generate control input value. While the feedback gain matrices for traditional linear optimal controller are independent of initial states, the number of control exercises, ν, and $K(i)$ matrices are dependent on initial states for temporal control systems. But, if the possible set of initial states is in $\mathcal{R}^1$ they are independent of the initial states. Effective deployment of temporal control requires that we know the range of initial state values and generate $K(i)$ matrices for each group. A sensitivity analysis is required to determine how many distinct matrices need to be stored.

In order to implement temporal control we require an operating system that supports scheduling control computations at specific time instants. The Maruti system developed at the University of Maryland is a suitable host for the implementation of temporal control [10, 8, 7]. In Maruti, all executions are scheduled in time and the time of execution can be modified dynamically, if so desired. This is in contrast with traditional cyclic executives often used in real-time systems, which have a fixed, cyclic operation and which are well suited only for the sampled data control systems operating in a static environment. It is the availability of the system such as Maruti that allows us to consider the notion of temporal control, in which time becomes an emergent property of the system.

5 Example

To illustrate the advantages of a temporal control scheme let us consider a simple example of rigid body satellite control problem [12]. The system state equations are as follows:

$$
\begin{aligned}
x(k+1) &= \begin{bmatrix} 0 & 1 \\ -1 & 2 \end{bmatrix} x(k) + \begin{bmatrix} 0 \\ 0.00125 \end{bmatrix} u(k) \\
y(k) &= \begin{bmatrix} 1 & 1 \end{bmatrix} x(k)
\end{aligned}
$$

where k represents the time index and one unit of time is the discretized subinterval of length $\Delta = 0.05$. The linear quadratic performance index J'_M in (5) is used here with the following parameters.
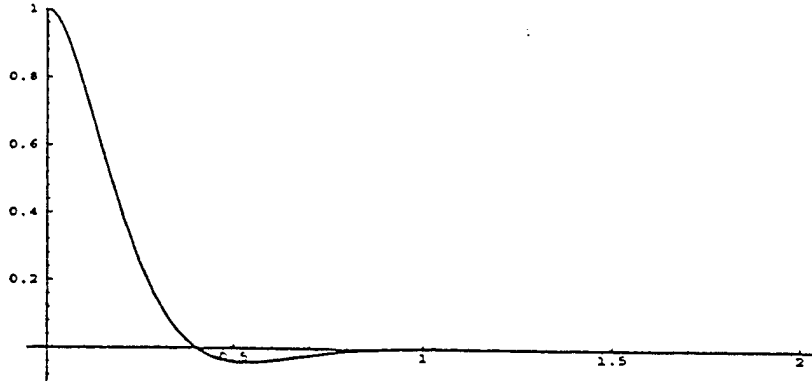
$$
Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}
$$

53

Figure 3: Optimal Linear Control with $\Delta = 0.05$.

$$
\begin{aligned}
R &= 0.0001 \\
\mu &= 0.02 \ \& \ 0.01 \\
M &= 40 \\
\Delta &= 0.05 \\
\epsilon_i &= 0.01, \quad i = 1, 2 \\
x(0) &= \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}
\end{aligned}
\tag{28}
$$

The objective of the control is to drive the satellite to the zero position and the desired goal state is $x_f = [0, \ 0]^T$. The terminal state constraint is $\mid x_i(40) \mid \leq \epsilon_i \quad i = 1, 2$. With the equal sampling interval $\Delta = 0.05$ and $M = 40$ the optimal linear feedback control of this system has cost function $J_M = 0.984678$ (without computational cost) and $J'_M = 1.784678$ (with computational cost) and is shown in Figure 3. The terminal state constraint is satisfied at $0.8sec$.

If we apply the temporal control scheme presented above to this problem with $\mu = 0.02$ we find that the optimal number of control changes for this example is 3 and $D_3^o = \{0, 2\Delta, 10\Delta\}$ with a cost $J'_M = 1.08388$. Note that the 40 step optimal linear feedback controller given above has a cost $J'_M = 1.784678$ when computation cost is considered. Table 1 shows how this optimal controller is obtained when we set $\nu_{max} = 7$. Figure 4(a) shows the system trajectory when this three-step optimal temporal controller is used to control the system. This trajectory satisfies the terminal state constraint at $0.8sec$ as well. Also, the maximum control input magnitudes, $\mid u \mid_{max}$, in both

n	D_ν^o	Cost(J'_M) with $\mu = 0.02$	Cost(J'_M) with $\mu = 0.01$
1	$\{0\}$	$4.63089 + \mu = 4.65089$	$4.63089 + \mu = 4.64089$
2	$\{0, 1\}$	$1.44603 + 2\mu = 1.48603$	$1.44603 + 2\mu = 1.46603$
3	$\{0, 2, 10\}$	$1.02388 + 3\mu = 1.08388$	$1.02388 + 3\mu = 1.05388$
4	$\{0, 2, 9, 11\}$	$1.02224 + 4\mu = 1.10224$	$1.02224 + 4\mu = 1.06224$
5	$\{0, 1, 3, 8, 11\}$	$0.996968 + 5\mu = 1.096968$	$0.996968 + 5\mu = 1.046968$
6	$\{0, 1, 3, 8, 11, 24\}$	$0.996746 + 6\mu = 1.116746$	$0.996746 + 6\mu = 1.056746$
7	$\{0, 1, 3, 8, 11, 23, 25\}$	$0.996745 + 7\mu = 1.136745$	$0.996745 + 7\mu = 1.066745$

Table 1: Calculating optimal temporal controllers.

controllers lie within the same bound $B = 50$, which may be another constraint on control.

The optimal temporal controller found with $\mu = 0.01$ has $\nu = 5$ and $D_5^o = \{0, \Delta, 3\Delta, 8\Delta, 11\Delta\}$ with a cost $J_M = 0.996968$. Note that this cost is even less than 1.01269 which is obtained from the optimal controller with equal sampling period $0.1sec$ and 20 control changes.

If we change control values only at three time instants with equal sampling period, $13M = 0.65sec$, the total cost incurred is 2.2823(without computational cost) on the time interval $[0, 2]$. The cost is more than twice that of our optimal temporal controller and the terminal state constraint is not satisfied even at the end of the controlling interval of $2.0sec$. Figure 4(b) clearly shows the advantages of using an optimal temporal controller over using an optimal controller of equidistant samplings. Their performances are noticeably different though both of them are changing controls at three time instants. It is clear that the optimal temporal control with three control changes performs almost the same as 40 step linear optimal controller does. This implies that enforcing the constant sampling rate throughout the entire controlling interval may simply waste computational power which otherwise could be used for other concurrent controlling tasks in critical systems.

Obtaining D_3^o for this example was simple since J_{40} has only one minimum over the entire set of possible D_3s on $[0, 40\Delta]$. Figure 5(a) and Figure 5(b) show that J_{40} has only one local(global) minimum at $D_3^o = \{0, 2\Delta, 10\Delta\}$. We got this optimal D_3 by doing steepest descent search with the starting point $D_3^{init} = \{0, \Delta, 10\Delta\}$ after searching for only three points, $\{0, \Delta, 10\Delta\}$, $\{0, 2\Delta, 10\Delta\}$, $\{0, 3\Delta, 10\Delta\}$. Also, Figure 5(a) shows that choosing n_1 has greater influence on the total cost than n_2 since the cost varies more radically along the n_1 axis in the figure. This means that the initial stage of the control needs more attention than the later stage in this linear control problem.

But, if we change one of the parameters of performance index function, R, from 0.0001 to 0.001 we get two local minima at $D_3^1 = \{0, \Delta, 2\Delta\}$ and $D_3^2 = \{0, 3\Delta, 19\Delta\}$, among which D_3^2 is the
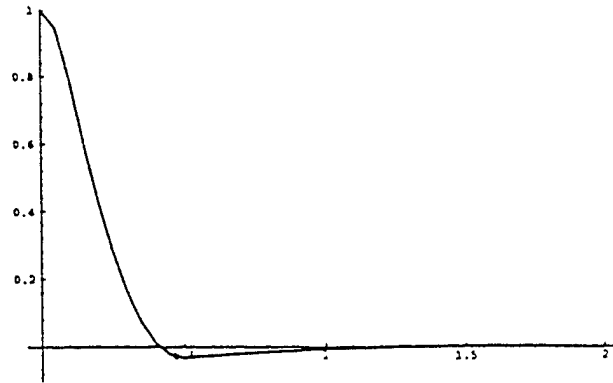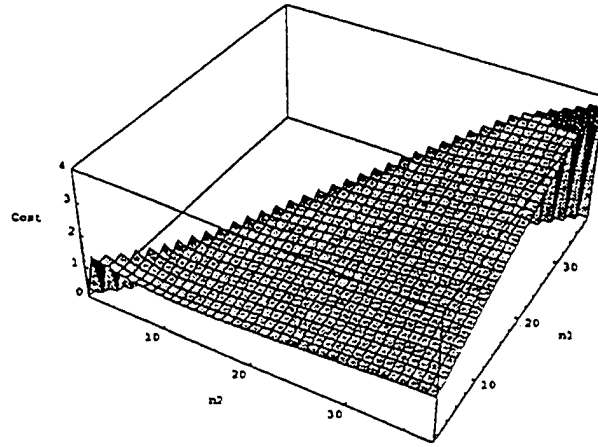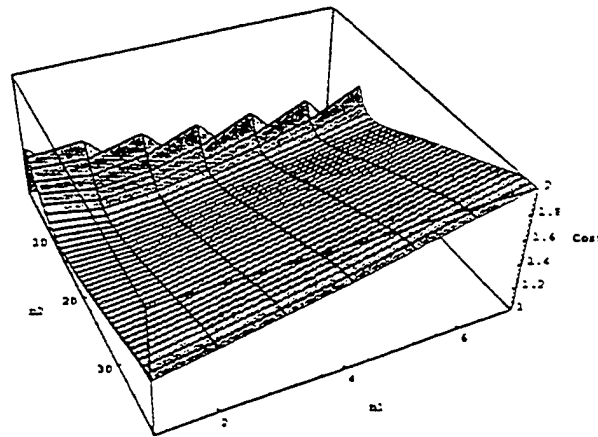
(a)
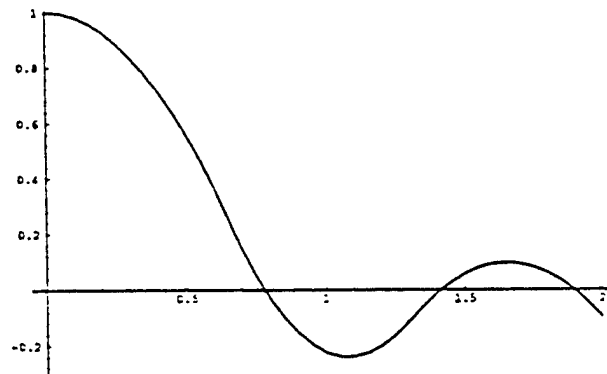
(b)

Figure 4: Control trajectories with 3 control changes. (a)Optimal temporal control with $D_3^o = \{0, 2\Delta, 10\Delta\}$. (b)Optimal linear control with 13Δ $(0.65sec)$ period.

(a)

(b)

Figure 5: Cost function distribution over (n_1, n_2). (a)Costs on D_3 space. (b)Costs near $D_3^o = \{0, 2\Delta, 10\Delta\}$.
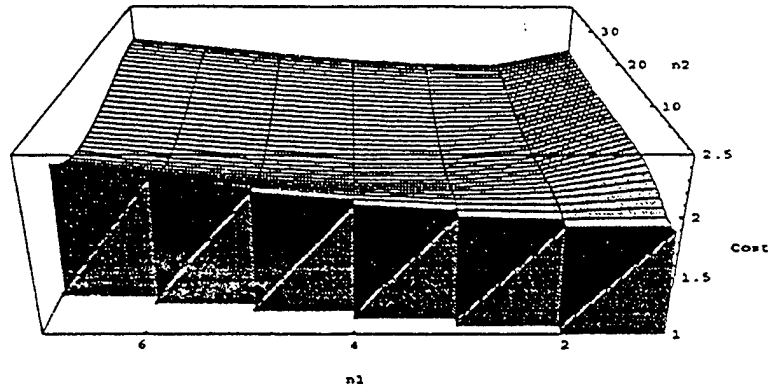
Figure 6: Costs near D_3^1 and D_3^2 with $R = 0.001$.

optimal one with less cost. Figure 6 shows this fact. In this case we need to use steepest descent search method at least twice with different search starting points to get an optimal solution. We implemented this steepest descent search algorithm in Mathematica and used it to generate D_ν^o for several examples by varying ν. For our examples of linear time invariant system control problems the number of local minima was not so large that we could efficiently apply this search method just a few times with different initial D_ν^{init}s to get a global minimum without doing an exhaustive search over the entire D_ν space.

6   Discussion

Employing the temporal control methodology in concurrent real-time embedded systems will have a significant impact on the way computational resources are utilized by control tasks. A minimal amount of control computations can be obtained for a given regulator by which we can achieve almost the same control performance compared to that of traditional controller with equal sampling period. This significantly reduces the CPU times for each controlling task and thus increases the number of real-time control functions which can be accommodated concurrently in one embedded system. Particularly, in a hierarchical control system if temporal controllers can be employed for lower level controllers the higher level controllers will have a great degree of flexibility in managing resource usages by adjusting computational requirements of each lower level controller. For example, in emergency situations the higher level controller may force the lower level controller to run as

infrequently as they possibly can (thus freeing computational resources for handling the emergency). In contrast, during normal operations the temporal control tasks may run as necessary, and the additional computation time can be used for higher level functions such as monitoring and planning, etc.

In addition, the method developed in Section 3.2, which calculates an optimal controller when control changing time instants are given, can be applied to the case in which the control computing time instants cannot be periodic. For example, when a small embedded controller is used to control several functions, it may be a lot better to design a temporal controller for each function such that the required computational resources are appropriately scheduled while retaining the required degree of control for each function.

7   Conclusion

In this paper we proposed a *temporal control* technique based on a new cost function which takes into account computational cost as well as state and input cost. In this scheme new control input values are defined at time instants which are not necessarily regularly spaced. For the linear control problem we showed that almost the same quality of control can be achieved while much less computations are used than in a traditional controller.

The proposed formulation of temporal control is likely to have a significant impact on the way concurrent embedded real-time systems are designed. In hierarchical control environment, this approach is likely to result in designs which are significantly more efficient and flexible than traditional control schemes. As it uses less computational resources, the lower level temporal controllers will make the resources available to the higher level controllers without compromising the quality of control.

References

[1] A. Belleisle. Stability of systems with nonlinear feedback through randomly time-varying delays. *IEEE Transactions on Automatic Control*, AC-20:67–75, February 1975.

[2] R. Bellman. *Adaptive Control Process: A Guided Tour*. Princeton,NJ: Princeton University Press, 1961.

[3] R. Bellman. Bellman special issue. *IEEE Transactions on Automatic Control*, AC-26, October 1981.

[4] P. Dorato and A. Levis. Optimal linear regulators: The discrete time case. *IEEE Transactions on Automatic Control*, AC-16:613–620, December 1971.

[5] A. Gosiewski and A. Olbrot. The effect of feedback delays on the performance of multivariable linear control systems. *IEEE Transactions on Automatic Control*, AC-25(4):729–734, August 1980.

[6] K. Hirai and Y. Satoh. Stability of a system with variable time delay. *IEEE Transactions on Automatic Control*, AC-25(3):552–554, June 1980.

[7] S. T. Levi, Satish K. Tripathi, Scott Carson, and Ashok K. Agrawala. The MARUTI hard real-time operating system. *ACM Symp. on Op. Syst. Principles, Op. Syst. Review*, 23(3), July 1989.

[8] Shem-Tov Levi and Ashok K. Agrawala. *Real Time System Design*. McGraw Hill, 1990.

[9] Z. Rekasius. Stability of digital control with computer interruptions. *IEEE Transactions on Automatic Control*, AC-31:356–359, April 1986.

[10] Manas Saksena, James da Silva, and Ashok K. Agrawala. *Design and Implementation of Maruti-II*, chapter 4. Prentice Hall, 1995. In *Advances in Real-Time Systems*, edited by Sang H. Son.

[11] K. G. Shin and H. Kim. Derivation and application of hard deadlines for real-time control systems. *IEEE Transactions on Systems, Man and Cybernetics*, 22(6):1403–1413, November 1992.

[12] G.S. Virk. *Digital Computer Control Systems*, chapter 4. McGraw Hill, 1991.

[13] K. Zahr and C. Slivinsky. Delay in multivariable computer controlled linear systems. *IEEE Transactions on Automatic Control*, pages 442–443, August 1974.

REPORT DOCUMENTATION PAGE

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	July 1995	Technical Report

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Designing Temporal Controls	N00014-91-C-0195 and DSAG-60-92-C-0055

6. AUTHOR(S)

Ashok K. Agrawala, Seonho Choi & Leyuan Shi

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
University of Maryland A.V. Williams Building College Park, Maryland 20742	CS-TR-3504 UMIACS-TR-95-81

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING / MONITORING AGENCY REPORT NUMBER
Honeywell Phillips Labs 3660 Technology Drive 3550 Aberdeen Ave. SE Minneapolis, MN 55418 Kirtland AFB, NM 87117-5776	

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Traditional control systems have been designed to exercise control at regularly spaced time instants. When a discrete version of the system dynamics is used, a constant sampling interval is assumed and a new control value is calculated and exercised at each time instant. In this paper we formulate a new control scheme, temporal control, in which we not only calculate the control value but also decide time instants when the new values are to be used. Taking a discrete, linear, time-invariant system, and a cost function which reflects a cost for computation of the control values, as an example, we show the feasibility of using this scheme. We formulate the temporal control scheme as a feedback scheme and, through a numerical example, demonstrate the significant reduction in cost through the use of temporal control.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Computing Methodologies	22 pages
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited

Scheduling an Overloaded Real-Time System *

Shyh-In Hwang, Chia-Mei Chen, and Ashok K. Agrawala

Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland, College Park, MD 20742

Abstract

The real-time systems differ from the conventional systems in that every task in the real-time system has a timing constraint. Failure to execute the tasks under the timing constraints may result in fatal errors. Sometimes, it may be impossible to execute all the tasks in the task set under their timing constraints. Considering a system with limited resources, one solution to handle the overload problem is to reject some of the tasks in order to generate a feasible schedule for the rest. In this paper, we consider the problem of scheduling a set of tasks without preemption in which each task is assigned criticality and weight. The goal is to generate an optimal schedule such that all of the critical tasks are scheduled and then the non-critical tasks are included so that the weight of rejected non-critical tasks is minimized. We consider the problem of finding the optimal schedule in two steps. First, we select a permutation sequence of the task set. Secondly, a pseudo-polynomial algorithm is proposed to generate an optimal schedule for the permutation sequence. If the global optimal is desired, all permutation sequences have to be considered. Instead, we propose to incorporate the simulated annealing technique to deal with the large search space. Our experimental results show that our algorithm is able to generate near optimal schedules for the task sets in most cases while considering only a limited number of permutations.

1 Introduction

Real-time computer systems are essential for all embedded applications, such as robot control, flight control, and medical instrumentation. In such systems, the computer is required to support the execution of applications in which the timing constraints of the tasks are specified by the physical system being controlled. The correctness of the system depends on the temporal correctness as well as the functional correctness of the tasks. Failure to satisfy the timing constraints can incur fatal errors. How to schedule the tasks so that their timing constraints are met is crucial to the proper operation of a real-time system.

As an example of an embedded system, let us consider the air defense system which monitors an air space continuously using radars. Whenever an intruder is identified, the embedded control system characterizes it and proceeds to initiate the responsive action in a timely manner. The temporal constraints for this phase of processing are different depending on the intruder, whether it is a missile, a fighter, a bomber, a dummy, etc. Such a system is designed to handle a number of intruders concurrently. If the processing requests exceed the capacity of the system, we expect the system to handle a set of the most significant intruders, and not any arbitrary set of intruders. This involves rejecting the processing of some real-time tasks based on their importance. In this paper, we consider the problem of creating a schedule for a set of tasks such that all critical tasks are scheduled, and then, among the non-critical tasks we select those which can be scheduled feasibly while maximizing the sum of the weights of selected non-critical tasks.

As all systems have finite resources, their ability to execute a set of tasks while meeting the temporal requirements is limited. Clearly, overload conditions may arise if more tasks have to be processed than the available set of resources can handle. Under such overload conditions, we have two choices. We may augment the resources available, or reject some tasks (or both). In [8], a technique was presented to handle transient overloads by taking advantage of redundant computing resources. Another permissible solution to this problem is to reject some of the tasks in order to generate a feasible schedule for the rest. Once a task is accepted by the system, the system should be able to finish it under its timing constraint. Some algorithms may have been shown to perform

well under low or moderate resource utilization. However, their performance degrades if the system is overloaded [2]. For example, the EDF algorithm has been shown to be optimal for a periodic task set [6]. If there exists a feasible schedule for the task set, EDF can come up with one. However, if the task set is not feasible, EDF may perform unsatisfactorily. The reason is that a task with urgent deadline may not be able to finish before its deadline. But, due to its urgent deadline, the task has a high priority to use the processor and thus keeps wasting the CPU time until the task expires after its deadline. The waste of CPU time may further prevent other tasks from meeting their deadlines. The other problem is that there is little control over which tasks will meet their deadlines and which will not.

For an overloaded system, how to select tasks for rejection on the basis of their importance becomes a significant issue. When the tasks have equal weight, an optimal schedule can be defined to be one in which the number of rejected tasks is minimized. In our previous study [3], we used a *super sequence* based scheduling algorithm to compute the optimal schedule for the tasks. In this paper, the criticality of the tasks are taken into consideration. Basically, if a task can not meet its deadline, it is rejected so that the CPU time would not be wasted. Secondly, we would like to schedule tasks such that the less important tasks may be rejected in favor of the more important tasks. We classify tasks into two categories: *critical* and *non-critical*. The critical tasks are crucial to the system such that they must not be rejected. The non-critical tasks are given weights to reflect their importance, and are allowed to be rejected. A schedule is *feasible* if all critical tasks in the task set are accepted and are guaranteed to meet their timing constraints. If there exists no feasible schedule for the task set, the task set is considered infeasible. The *loss* of a schedule is defined to be the sum of the weights of the rejected non-critical tasks. A schedule is *optimal* if it is feasible and the loss of the schedule is minimum.

We first propose a Permutation Scheduling Algorithm (PSA) to generate an optimal schedule for a permutation, which is a well defined ordering of tasks. When it comes to scheduling a task set of n tasks, in the worst case there might be up to $n!$ permutations to consider. We propose a Set Scheduling Algorithm (SSA) which incorporates the *simulated annealing* technique [9] to deal with the large search space of permutations. PSA is invoked by SSA to compute the optimal schedule for

each permutation. Taking the feedback from the schedulability and loss of the schedule generated by PSA, SSA is able to control the progress of search for an optimal schedule for the task set. Our experimental results show that SSA is able to generate feasible schedules for task sets consisting of 100 tasks with success ratios no less than 98% and loss ratios less than 10% for most cases while searching less than 5,000 permutations. For each permutation, the average number of schedules computed to generate an optimal schedule by PSA, which is invoked by SSA, is usually less than 500. The SSA algorithm can be considered efficient in dealing with the exponential search space for coming up with a satisfactorily near optimal schedule.

In the following section, we define the scheduling problem. In section 3, we present the idea about how to schedule a permutation. In section 4, we incorporate the technique of simulated annealing and discuss how to schedule a task set. In section 5, the results of our experiments are presented, which is followed by our conclusion.

2 The Problem

A task set is represented as $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$. A task τ_i can be characterized as a record of (r_i, c_i, d_i, w_i), representing the ready time, computation time, deadline, and criticality of the ith task. Time is expressed as a real number. A task can not be started before its ready time. Once started, the task must use the processor without preemption for c_i time units, and be finished by its deadline. If a task is very important for the system such that rejection of the task is not allowed, w_i is set to be CRITICAL. Otherwise, w_i is assigned an integral value to indicate its importance, and is subject to rejection if necessary. A *permutation sequence*, or simply abbreviated to a *permutation*, is an ordered sequence of tasks in the task set. Scheduling is a process of binding starting times to the tasks such that each task executes according to the schedule. Note that a non-preemptive schedule on a single processor implies a sequence for the execution of tasks. For the convenience of our discussion, we hereafter use a sequence to represent the schedule in the context. A permutation is denoted by $\mu = \langle \tau_1, ..., \tau_n \rangle$, where τ_i is the ith task in the permutation. A prefix of a permutation is denoted by $\mu_k = \langle \tau_1, ..., \tau_k \rangle$.

To schedule a task set, we need to take into consideration the possible permutations in the task set. We first consider an algorithm for scheduling a permutation. The finish time of a schedule is the finish time of the last task in the schedule. Let $S_k(t)$ denote a schedule of μ_k with finish time no more than t. We use $W(S_k(t))$ to represent the weight of $S_k(t)$, which is the sum of the weights of non-critical tasks in the schedule. A feasible schedule of μ_k is defined as follows:

Definition: $S_k(t)$, $1 \leq k \leq n$, is a *feasible* schedule of μ_k at t, if and only if:

1. $S_k(t)$ is a subsequence of μ_k,

2. the finish time of $S_k(t)$ is less than or equal to t, and

3. all critical tasks in μ_k are included in $S_k(t)$.

An optimal schedule of μ_k is defined as follows:

Definition: $\sigma_k(t)$ is an *optimal* schedule of μ_k at t, if and only if:

1. $\sigma_k(t)$ is a feasible schedule of μ_k, and

2. for any feasible schedule $S_k(t)$ of μ_k, $W(\sigma_k(t)) \geq W(S_k(t))$.

In other words, an optimal schedule is a feasible schedule with minimum loss. There are possibly more than one optimal schedules for μ_k with finish time less than or equal to t. We donote by $\Sigma_k(t)$ the set of all of the optimal schedules for μ_k at t. Hence, if $S_k(t) \in \Sigma_k(t)$, $S_k(t)$ is an optimal schedule for μ_k at t.

The scheduling problem considered here is NP-complete. To prove that, its related *decision problem*, which is defined to be computing a feasible schedule with loss no more than a given bound, can be easily shown to be NP-complete. This can be done by restricting to PARTITION problem [1] by setting $r_i = 0, w_i = c_i, d_i = \frac{1}{2} \sum_{j=1}^{n} c_j$, for $1 \leq i \leq n$.

3   Scheduling a Permutation

We consider the problem of finding an optimal schedule for the task set in two steps – select a permutation, and find an optimal schedule for the permutation. The methodology is presented in Figure 1.

Loop 1: Choose a permutation μ of Γ

    Loop 2: for μ_k, $k = 1, 2, \ldots, n$

        Loop 3: compute $\sigma_k(t)$

Figure 1: Methodology

Clearly, to find the optimal schedule for the task set, all possible permutations have to be considered. How to search the permutations will be addressed in section 4. In Loop 3, optimal schedules for μ_k are computed at some time instants. Next, we discuss how to compute $\sigma_k(t)$ for a given t in the following, and then discuss how to determine the time instants for μ_k.

3.1 Computing $\sigma_k(t)$

We use dynamic programming to compute $\sigma_k(t)$ based on $\sigma_{k-1}(t')$, with $t' \leq t$. The criticality of τ_k plays an important role in computing $\sigma_k(t)$.

If τ_k is a critical task, we have to schedule it, possibly at the cost of rejecting some of the non-critical tasks. Hence, $\sigma_k(t) = S_{k-1}(t') \oplus \tau_k$, for some schedule $S_{k-1}(t')$, where $\oplus$ means concatenation of the sequence and the task. The finish time of $S_{k-1}(t')$ must be no more than $t - c_k$ in order to accommodate τ_k, which leads to $t' \leq t - c_k$. The best candidate could be $\sigma_{k-1}(t - c_k)$. Hence,

$$\sigma_k(t) = \sigma_{k-1}(t - c_k) \oplus \tau_k, \tag{1}$$

which can be seen in Figure 2. Note that $\sigma_k(t)$ only exists for a *proper* range of t. That is, $\sigma_k(t)$ is infeasible when t is beyond the proper range, e.g., $t < \tau_k + c_k$, or if $\sigma_{k-1}(t - c_k)$ is infeasible. The range would be considered in details later.

If τ_k is non-critical, our concern is to obtain as large a weight for the schedule as possible, while the critical tasks accepted previously must be kept in the schedule. Computation of $\sigma_k(t)$ is based

68
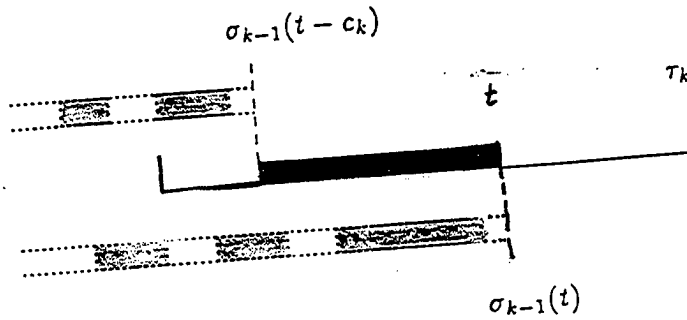
Figure 2: Scheduling for τ_k

upon the choice between either including τ_k or not. That is,

$$\sigma_k(t) = \begin{cases} \sigma_{k-1}(t - c_k) \oplus \tau_k & \text{or} \\ \sigma_{k-1}(t) \end{cases} \tag{2}$$

which can be seen in Figure 2. The factors for making the choice are the feasibility and the weights of the two candidate schedules. That is, the chosen schedule has to be feasible in the first place, and has a weight more than or equal to the other.

3.2 Time Instants for Computing $\sigma_k(t)$

From Equations 1 and 2, the computation of $\sigma_k(t)$ is based on the results of $\sigma_{k-1}(t)$ and $\sigma_{k-1}(t-c_k)$. We do not need to look for all possible values for t. We can get the idea about how to determine the time instants t by a simple example in Figure 3. The ready times, computation times, deadlines, and weights are given to the tasks in $\mu_3 = \langle \tau_1, \tau_2, \tau_3 \rangle$.

The following schedules for μ_3 can be easily verified.

$$\sigma_3(t) = \text{INFEASIBLE}$$
$$\sigma_3(t) = \langle \tau_3 \rangle$$
$$\sigma_3(t) = \langle \tau_2, \tau_3 \rangle$$
$$\sigma_3(t) = \langle \tau_1, \tau_3 \rangle$$

$$W(\sigma_3(t)) = 0$$
$$W(\sigma_3(t)) = 5$$
$$W(\sigma_3(t)) = 10$$

for $t < 6$
for $6 \le t < 7.5$
for $7.5 \le t < 9$
for $9 \le t$
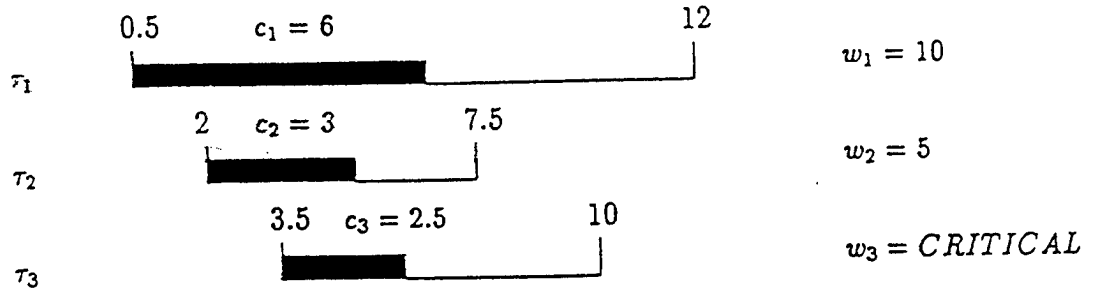
Figure 3: $\mu_3 = \langle \tau_1, \tau_2, \tau_3 \rangle$

In general, there exist a number of subranges in each of which the schedules are exactly identical, which are illustrated in Figure 4. We only need to compute the schedules at the time instants which delimit the subranges, i.e., $6, 7.5$, and 9. We call these time instants *scheduling points*. The scheduling points can be determined by the timing characteristics of the tasks.
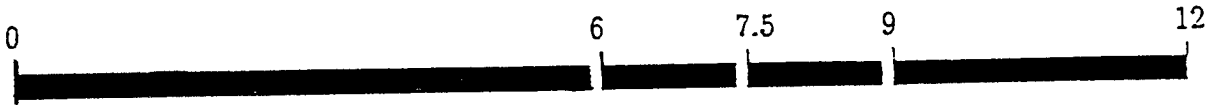
Figure 4: Identical subranges

3.3   Definition of Scheduling Points

We denote the jth scheduling point for μ_k by $\lambda_{k,j}$, and call j the *index* of $\lambda_{k,j}$. Hence, $\sigma_k(\lambda_{k,j})$ denotes an optimal schedule for μ_k at the scheduling point $\lambda_{k,j}$. Let v_k be the total number of scheduling points at which we need to schedule μ_k. For simplicity, λ_k denotes the set of $\lambda_{k,1}, \lambda_{k,2}, \ldots, \lambda_{k,v_k}$, and σ_k the set of $\sigma_k(\lambda_{k,1}), \sigma_k(\lambda_{k,2}), \ldots, \sigma_k(\lambda_{k,v_k})$. The scheduling points are defined as follows.

Definition: The set of scheduling points, λ_k, is *complete* if and only if:

1. for any $t < \lambda_{k,1}$, $\Sigma_k(t)$ is empty,

2. for any $\lambda_{k,j} \leq t < \lambda_{k,j+1}$, for $j = 1, \ldots, v_k - 1$, $\sigma_k(\lambda_{k,j}) \in \Sigma_k(t)$, and

3. for any $t \geq \lambda_{k,v_k}$, $\sigma_k(\lambda_{k,v_k}) \in \Sigma_k(t)$.

Note that $\Sigma_k(t)$ being empty means that there is no feasible schedule with finish time less than or equal to t. And also remember that $\sigma_k(\lambda_{k,j}) \in \Sigma_k(t)$ means that $\sigma_k(\lambda_{k,j})$ is an optimal

schedule for μ_k at t. The completeness of scheduling points indicates that all of the optimal schedules at the positive real time domain can be represented by the optimal schedules computed at the scheduling points. In addition, the set of scheduling points, λ_k, is *minimum*, if and only if $W(\sigma_k(\lambda_{k,j})) < W(\sigma_k(\lambda_{k,j+1}))$, for any $1 \leq j \leq v_k - 1$. This ensures that there does not exist any redundant scheduling point which, if removed, does not violate the completeness of the scheduling points. The sets of scheduling points that we will discuss are complete and minimum.

3.4 An Example for Deriving Scheduling Points

The values of λ_k depend on the temporal relations between τ_k and λ_{k-1}. The example in Figure 5 is used to illustrate the relations. We only describe the idea of deriving scheduling points by the example, and will discuss in more details later. Assume that there are 5 scheduling points for μ_{k-1}, and we consider to compute σ_k based on σ_{k-1}. The current task, τ_k, may be critical or non-critical.

scheduling points for μ_{k-1} :
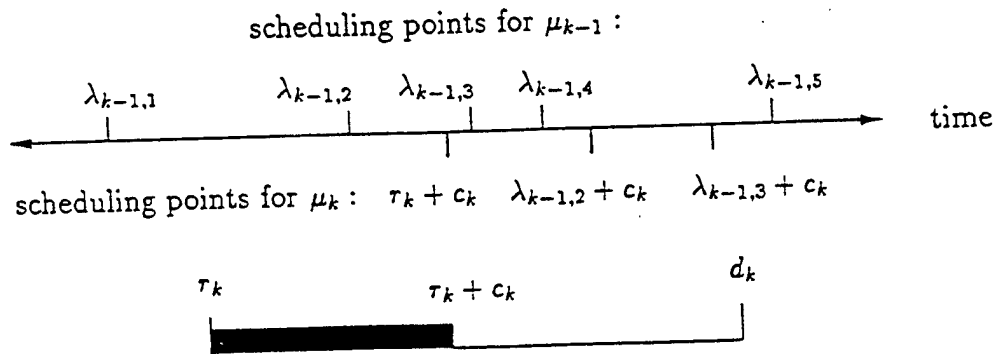
Figure 5: Scheduling Points

First, let us assume that τ_k is critical, which means that τ_k must be the last task in any feasible schedules for μ_k. A schedule for μ_k is thus a schedule for μ_{k-1} concatenated by τ_k. Hence, the optimal schedules for μ_k can be computed by appending τ_k to $\sigma_{k-1}(j)$, $j = 1, \ldots, v_{k-1}$. One restriction is that τ_k must be able to execute during its time window, from τ_k to d_k. Hence, the scheduling points are $\lambda_{k-1,j} + c_k$, $j = 1, \ldots, v_{k-1}$, subject to the timing constraint of τ_k. In the example, because $\tau_k > \lambda_{k-1,1}$, the first scheduling point is $\lambda_{k,1} = \tau_k + c_k$. The first and the rest scheduling points are expressed in Equations 3-5. Notice that $\lambda_{k-1,4} + c_k > d_k$. Hence, there are

71

only 3 scheduling points for μ_k.

$$\lambda_{k,1} = r_k + c_k \qquad \text{and} \quad \sigma_k(\lambda_{k,1}) = \sigma_{k-1}(\lambda_{k-1,1}) \oplus \tau_k \tag{3}$$

$$\lambda_{k,2} = \lambda_{k-1,2} + c_k \quad \text{and} \quad \sigma_k(\lambda_{k,2}) = \sigma_{k-1}(\lambda_{k-1,2}) \oplus \tau_k \tag{4}$$

$$\lambda_{k,3} = \lambda_{k-1,3} + c_k \quad \text{and} \quad \sigma_k(\lambda_{k,3}) = \sigma_{k-1}(\lambda_{k-1,3}) \oplus \tau_k \tag{5}$$

On the other hand, let us assume that τ_k is non-critical. As a non-critical task, τ_k is not necessarily included in the schedule of μ_k. Whether to include τ_k or not depends on how much weight may be gained by including τ_k. If τ_k is included in the schedules, the new possible scheduling points for μ_k are expressed in Equations 6-8.

$$\lambda'_{k,1} = r_k + c_k \qquad \text{and} \quad \sigma'_k(\lambda'_{k,1}) = \sigma_{k-1}(\lambda_{k-1,1}) \oplus \tau_k \tag{6}$$

$$\lambda'_{k,2} = \lambda_{k-1,2} + c_k \quad \text{and} \quad \sigma'_k(\lambda'_{k,2}) = \sigma_{k-1}(\lambda_{k-1,2}) \oplus \tau_k \tag{7}$$

$$\lambda'_{k,3} = \lambda_{k-1,3} + c_k \quad \text{and} \quad \sigma'_k(\lambda'_{k,3}) = \sigma_{k-1}(\lambda_{k-1,3}) \oplus \tau_k \tag{8}$$

If τ_k is not included, the scheduling points for μ_k are $\lambda_{k-1,j}, j = 1, \ldots, v_{k-1}$. The scheduling points for μ_k can be derived by, first, merging and sorting λ'_k and λ_{k-1}, which gives

$$\lambda_{k-1,1}, \lambda_{k-1,2}, \lambda'_{k,1}, \lambda_{k-1,3}, \lambda_{k-1,4}, \lambda'_{k,2}, \lambda'_{k,3}, \lambda_{k-1,5}. \tag{9}$$

Then, the resultant array of scheduling points should follow the rule that the weights of the optimal schedules at the scheduling points in the resultant array in Equation 9 should be strictly increasing. We remove any scheduling point if necessary.

3.5 Deriving Scheduling Points

By the example illustrated in Figure 5, λ_k can be derived from λ_{k-1} and τ_k. Note that a scheduling point indicates the finish time of a schedule. If we want to append τ_k to $\sigma_{k-1}(\lambda_{k-1,j})$, τ_k can not be started before $\lambda_{k-1,j}$. This implies that λ_k can be determined by the temporal relations between λ_{k-1}, the finish times of σ_k, and the start time of τ_k. Specifically, we need to explore the temporal relations between the earliest start time, r_k, the latest start time, $d_k - c_k$, of τ_k, and the lower and

72

upper bounds to be defined below. We define the lower bound $L_{k-1} \equiv \lambda_{k-1,1}$, and the upper bound $U_{k-1} \equiv \lambda_{k-1,v_{k-1}}$. In particular, they have the following meanings.

L_{k-1}: the largest time instant such that there is no feasible schedule for μ_{k-1} with finish time less than L_{k-1}.

U_{k-1}: the least time instant such that the optimal schedule for μ_{k-1} with finish time greater than U_{k-1} can be $\sigma_{k-1}(\lambda_{k-1,v_{k-1}})$.

The six possible temporal relations in Equations 10–15 can be used to determine λ_k.

$$d_k - c_k < L_{k-1} \leq U_{k-1} \tag{10}$$

$$\tau_k \leq L_{k-1} \leq d_k - c_k < U_{k-1} \tag{11}$$

$$L_{k-1} < \tau_k \leq d_k - c_k < U_{k-1} \tag{12}$$

$$\tau_k \leq L_{k-1} \leq U_{k-1} \leq d_k - c_k \tag{13}$$

$$L_{k-1} < \tau_k \leq U_{k-1} \leq d_k - c_k \tag{14}$$

$$L_{k-1} \leq U_{k-1} < \tau_k \tag{15}$$

The temporal relations are illustrated in Figure 6, and can be summarized in three cases. The method for constructing scheduling points according to the temporal relations is discussed next. The correctness of the method, i.e., the completeness and minimization of the scheduling points , is verified later.

3.5.1 τ_k is Critical

The task τ_k must be the last task in any feasible schedule of μ_k. Remember that $\sigma_k(t)$ can be computed by Equation 1. In the following, we discuss how to derive the scheduling points for the three cases. The readers may refer to the algorithm in section 3.7 for details.

Case 1 $d_k - c_k < L_{k-1}$: μ is not feasible. Remember that there exists no feasible schedule for μ_k with finish time less than L_{k-1}, due to the completeness of scheduling points, and that $d_k - c_k$ is the latest start time for τ_k. Hence, μ_k is not feasible, and thus the whole permutation, μ, is not feasible.

73
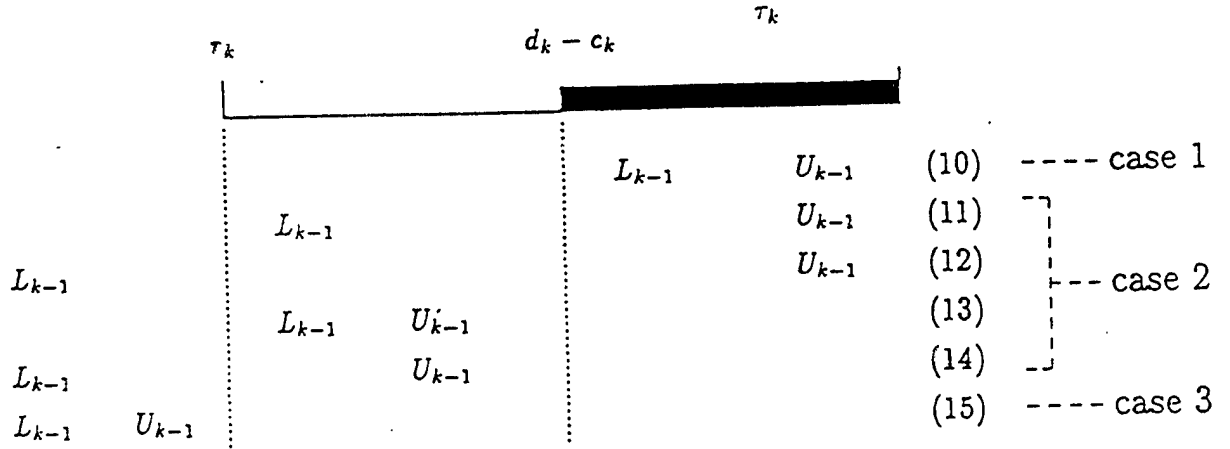
Figure 6: Temporal relations

Case 2 ($\tau_k \leq L_{k-1} \leq d_k - c_k$) or ($L_{k-1} < \tau_k \leq U_{k-1}$) : The scheduling points for μ_k is the set of $\lambda_{k-1,j} + c_k$, $j = 1, \ldots, v_{k-1}$, subject to the constraints that τ_k must start after τ_k, and finish before d_k. Specifically, λ_k can be derived by Equations 16 and 17.

$$\lambda_{k,1} = max(\lambda_{k-1,1} + c_k, \tau_k + c_k) \tag{16}$$

Let J_{min} and J_{max} denote the smallest and the largest integers of j satisfying $\lambda_{k,1} < \lambda_{k-1,j} + c_k \leq d_k$. The rest of the scheduling points can be computed by

$$\lambda_{k,i} = \lambda_{k-1,j} + c_k, \quad where \; J_{min} \leq j \leq J_{max} \; and \; i = j - J_{min} + 2 \tag{17}$$

Note that $v_k = J_{max} - J_{min} + 2$. The example given in Figure 5 falls in this case.

Case 3 $U_{k-1} < \tau_k$: there is only one scheduling point. Since τ_k is the earliest start time for τ_k, the only scheduling point is $\tau_k + c_k$.

3.5.2 τ_k is Non-critical

Remember that $\sigma_k(t)$ can be computed by Equation 2. The non-critical task τ_k is not necessarily included in the schedule for μ_k. Whether to include τ_k or not depends on how much weight may be gained by including τ_k. Let us consider the three cases.

Case 1 $d_k - c_k < L_{k-1}$: do nothing. The latest start time of τ_k is less than the lower bound, L_{k-1}; hence, τ_k can not be included in any feasible schedule. The scheduling points and schedules for μ_{k-1} remain the same as the scheduling points and schedules for μ_k. In our implementation, to save time and space, λ_{k-1} and λ_k use the same memory spaces; also, σ_{k-1} and σ_k use the same memory spaces. So now $\lambda_k = \lambda_{k-1}$ and $\sigma_k = \sigma_{k-1}$.

Case 2 $(r_k \leq L_{k-1} \leq d_k - c_k)$ or $(L_{k-1} < r_k \leq U_{k-1})$: If τ_k is included, the new *possible* scheduling points for μ_k is the set of $\lambda_{k-1,j} + c_k$, $j = 1, \ldots, v_{k-1}$, subject to the constraints that τ_k must start after r_k, and finish before d_k. Specifically, the new possible scheduling points , λ'_k, can be derived by Equations 18 and 19.

$$\lambda'_{k,1} = max(\lambda_{k-1,1} + c_k, r_k + c_k) \tag{18}$$

Let J_{min} and J_{max} denote the smallest and the largest integers of j satisfying $\lambda'_{k,1} < \lambda_{k-1,j} + c_k \leq d_k$. The rest of the scheduling points are

$$\lambda'_{k,i} = \lambda_{k-1,j} + c_k, \quad where \ J_{min} \leq j \leq J_{max} \ and \ i = j - J_{min} + 2 \tag{19}$$

If τ_k is not included, the scheduling points for μ_k are the old ones for μ_{k-1}; i.e.,

$$\lambda_{k-1,j}, \ j = 1, \ldots, v_{k-1}. \tag{20}$$

It is worth mentioning that some optimal schedules may include τ_k, and some may not. The scheduling points, λ_k, can be derived by the following two steps.

1. Merge and sort the two arrays of scheduling points, λ'_k and λ_{k-1}, in Equations 18-20.

2. The resultant array of scheduling points should follow the rule that the weights of the optimal schedules at the scheduling points should be strictly increasing. We remove any scheduling point that has a smaller weight than that of its preceding scheduling point in the array.

The example given in Figure 5 falls in this case.

Case 3 $U_{k-1} < r_k$: add one more scheduling point. The earliest start time of τ_k is greater than the upper bound, U_{k-1}; hence, the new scheduling point is $r_k + c_k$. The weight of the optimal schedule computed at this scheduling point is $W(\sigma_{k-1}(\lambda_{k-1,v_{k-1}})) + w_k$, which is larger than

$W'(\sigma_{k-1}(\lambda_{k-1,v_{k-1}}))$. So this scheduling point must be included to make the set of scheduling points for μ_k complete. Note again that the scheduling points and schedules for μ_{k-1} remain unchanged as the scheduling points and schedules for μ_k; i.e., $\lambda_{k,j} = \lambda_{k-1,j}$ and $\sigma_k(\lambda_{k,j}) = \sigma_{k-1}(\lambda_{k-1,j})$, for $j = 1, \ldots, v_{k-1}$. However, $\lambda_{k,v_k} = r_k + c_k$ and $\sigma_k(\lambda_{k,v_k}) = \sigma_{k-1}(\lambda_{k-1,v_{k-1}}) \oplus \tau_k$, where $v_k = v_{k-1} + 1$.

3.6 Completeness and Minimization of Scheduling Points

We would like to show that the sets of scheduling points derived in the three cases are complete and minimum. Note that cases 1 and 3 are special cases, and are not difficult to verify. Hence, we will only briefly discuss case 2. If τ_k is critical, we would like to show that If λ_{k-1} is complete and minimum, λ_k derived by Equations 16 and 17 is also complete and minimum.

Condition 1 of completeness: Due to the completeness of λ_{k-1}, $\Sigma_{k-1}(t)$ is empty when $t < \lambda_{k-1,1}$. Equivalently, $\Sigma_{k-1}(t - c_k)$ is empty when $t < \lambda_{k-1,1} + c_k$. According to Equation 1, $\sigma_k(t) = \sigma_{k-1}(t - c_k) \oplus \tau_k$. Hence, $\sigma_k(t)$ does not exist when $t < \lambda_{k-1,1} + c_k$. On the other hand, since τ_k is critical, $\sigma_k(t)$ does not exist when $t < r_k + c_k$, which is the earliest finish time of τ_k. Therefore, $\Sigma_k(t)$ is empty when $t < \lambda_{k,1}$. This shows that condition 1 of the definition of completeness is satisfied.

Condition 2 of completeness: Due to the completeness of λ_{k-1}, $\sigma_{k-1}(\lambda_{k-1,j}) \in \Sigma_{k-1}(t)$, for any $\lambda_{k-1,j} \le t < \lambda_{k-1,j+1}$. By Equation 1, $\sigma_{k-1}(\lambda_{k-1,j}) \oplus \tau_k$ is an optimal schedule at $\lambda_{k-1,j} + c_k$ for μ_k. Hence, $\sigma_{k-1}(\lambda_{k-1,j}) \oplus \tau_k \in \Sigma_k(t)$, for $\lambda_{k-1,j} + c_k \le t < \lambda_{k-1,j+1} + c_k$. By Equation 17, $\lambda_{k,i} = \lambda_{k-1,j} + c_k$, for $i = j - J_{min} + 2$, which indicates that $\sigma_k(\lambda_{k,i}) = \sigma_{k-1}(\lambda_{k-1,j}) \oplus \tau_k$. Besides, $\lambda_{k,i+1} = \lambda_{k-1,j+1} + c_k$, for $i + 1 = j + 1 - J_{min} + 2$, by Equation 17. Therefore, $\sigma_k(\lambda_{k,i}) \in \Sigma_k(t)$, for $\lambda_{k,i} \le t < \lambda_{k,i+1}$. This shows that condition 2 of the definition of completeness is satisfied.

Condition 3 of completeness: We know that $v_k = J_{max} - J_{min} + 2$. By Equation 17, $\lambda_{k,v_k} = \lambda_{k-1,J_{max}} + c_k$, which indicates that $\sigma_k(\lambda_{k,v_k}) = \sigma_{k-1}(\lambda_{k-1,J_{max}}) \oplus \tau_k$. Due to the completeness of λ_{k-1}, $\sigma_{k-1}(\lambda_{k-1,J_{max}}) \in \Sigma_{k-1}(t)$, for $\lambda_{k-1,J_{max}} \le t < \lambda_{k-1,J_{max}+1}$, or just $\lambda_{k-1,J_{max}} \le t$ if $J_{max} = v_{k-1}$. By Equation 1, $\sigma_{k-1}(\lambda_{k-1,J_{max}}) \oplus \tau_k$ is an optimal schedule at $\lambda_{k-1,J_{max}} + c_k$ for μ_k. Hence, $\sigma_{k-1}(\lambda_{k-1,J_{max}}) \oplus \tau_k \in \Sigma_k(t)$, for $\lambda_{k-1,J_{max}} + c_k \le t$. Note that the range of

$t < \lambda_{k-1,J_{max}+1} + c_k$ is removed. Because J_{max} is the largest integer of j satisfying $\lambda_{k-1,j} + c_k \leq d_k$, the schedule $\sigma_{k-1}(\lambda_{k-1,J_{max}+1}) \oplus \tau_k$ would not be feasible. Since $\sigma_k(\lambda_{k,v_k}) = \sigma_{k-1}(\lambda_{k-1,J_{max}}) \oplus \tau_k$, $\sigma_k(\lambda_{k,v_k}) \in \Sigma_k(t)$ for $\lambda_{k,v_k} \leq t$. This shows that condition 3 of the definition of completeness is satisfied.

Minimization: By Equation 1, $W(\sigma_k(t)) = W(\sigma_{k-1}(t - c_k) \oplus \tau_k) = W(\sigma_{k-1}(t - c_k))$, since a critical task has no weight. Because λ_{k-1} is minimum, $W(\sigma_{k-1}(\lambda_{k-1,j})) < W(\sigma_{k-1}(\lambda_{k-1,j+1}))$, for any $1 \leq j \leq v_{k-1} - 1$. That is, $W(\sigma_{k-1}(\lambda_{k-1,j}) \oplus \tau_k) < W(\sigma_{k-1}(\lambda_{k-1,j+1}) \oplus \tau_k)$, for any $1 \leq j \leq v_{k-1} - 1$. By Equations 16 and 17, $W(\sigma_k(\lambda_{k-1,j} + c_k)) < W(\sigma_k(\lambda_{k-1,j+1} + c_k))$, and thus $W(\sigma_k(\lambda_{k,i})) < W(\sigma_k(\lambda_{k,i+1}))$, for any $1 \leq i \leq v_k - 1$. This shows that λ_k is minimum.

If τ_k is non-critical, τ_k may be included or not included in the optimal schedules for μ_k. Assuming that τ_k is not included in any of the optimal schedules, $\lambda_k = \lambda_{k-1}$ is complete, since λ_{k-1} is complete. However, including τ_k may gain some more weight, so we also need to consider the schedules including τ_k. If τ_k is included in the optimal schedules, λ_k' derived by Equations 18 and 19 is the complete set of scheduling points for the optimal schedules including τ_k, by the same reason described for the critical task. Hence, it is sufficient to construct the complete set of λ_k by selecting from λ_k' and λ_{k-1}. Since whether to include τ_k or not does not affect the feasibility of the schedules, we only need to consider the weights of the optimal schedules. A complete set of scheduling points indicates that the weights of the optimal schedules at these scheduling points should be non-decreasing. Furthermore, a complete and minimum set of scheduling points indicates that the weights of the optimal schedules at these scheduling points should be strictly increasing. Hence, we can merge and sort the two arrays of λ_k' and λ_{k-1}, and remove any scheduling point that has a smaller weight than that of its preceding scheduling point in the array. The resultant scheduling points is thus complete and minimum.

3.7 The Permutation Scheduling Algorithm (PSA)

Algorithm <u>PSA</u>:

Input: a permutation sequence $\mu = \langle \tau_1, \tau_2, \ldots, \tau_n \rangle$

Output: an optimal schedule $\sigma_n(\lambda_{n,v_n})$

Initialization: $v_0 = 1$; $\lambda_{0,1} = 0$; $\sigma_0(\lambda_{0,1}) = \langle\rangle$; $W(\sigma_0(\lambda_{0,1})) = 0$

for $k = 1$ to n

    when $\boxed{\tau_k \text{ is critical}}$

        <u>case 1</u> $(d_k - c_k < L_{k-1})$: (μ is not feasible)

            exit

        <u>case 2</u> $(\tau_k \leq L_{k-1} \leq d_k - c_k)$ or $(L_{k-1} < \tau_k \leq U_{k-1})$:

            Computation for the first scheduling point:

$$\lambda_{k,1} = max(\lambda_{k-1,1} + c_k, \tau_k + c_k)$$

            $j = 1$ if $\lambda_{k-1,1} > \tau_k$; otherwise, j is the greatest integer such that $\lambda_{k-1,j} \leq \tau_k$

$$\sigma_k(\lambda_{k,1}) = \sigma_{k-1}(\lambda_{k-1,j}) \oplus \tau_k$$

$$W(\sigma_k(\lambda_{k,1})) = W(\sigma_{k-1}(\lambda_{k-1,j}))$$

            Loop: $j = J_{min}$ to J_{max}, where J_{min} and J_{max} denote the smallest and the largest

                    integers of j satisfying $\lambda_{k,1} < \lambda_{k-1,j} + c_k \leq d_k$.

$$i = j - J_{min} + 2$$

$$\lambda_{k,i} = \lambda_{k-1,j} + c_k$$

$$\sigma_k(\lambda_{k,i}) = \sigma_{k-1}(\lambda_{k-1,j}) \oplus \tau_k$$

$$W(\sigma_k(\lambda_{k,i})) = W(\sigma_{k-1}(\lambda_{k-1,j}))$$

$$v_k = J_{max} - J_{min} + 2$$

        <u>case 3</u> $(U_{k-1} < \tau_k)$: (only one scheduling point)

$$\lambda_{k,1} = \tau_k + c_k$$

$$\sigma_k(\lambda_{k,1}) = \sigma_{k-1}(\lambda_{k-1,v_{k-1}}) \oplus \tau_k$$

$$W(\sigma_k(\lambda_{k,1})) = W(\sigma_{k-1}(\lambda_{k-1,v_{k-1}}))$$

$$v_k = 1$$

when $\boxed{\tau_k \text{ is non-critical}}$

 case 1 $(d_k - c_k < L_{k-1})$: (scheduling points and schedules remain the same)

  /* Do nothing; τ_k cannot be included in any feasible schedule */

  /* Hence, $\lambda_k = \lambda_{k-1}$ and $\sigma_k = \sigma_{k-1}$ */

 case 2 $(\tau_k \leq L_{k-1} \leq d_k - c_k)$ or $(L_{k-1} < \tau_k \leq U_{k-1})$:

  Computation for the first new possible scheduling point:

$$\lambda'_{k,1} = max(\lambda_{k-1,1} + c_k, \tau_k + c_k)$$

  $j = 1$ if $\lambda_{k-1,1} > \tau_k$; otherwise, j is the greatest integer such that $\lambda_{k-1,j} \leq \tau_k$

$$\sigma'_k(\lambda'_{k,1}) = \sigma_{k-1}(\lambda_{k-1,j}) \oplus \tau_k$$

$$W(\sigma'_k(\lambda'_{k,1})) = W(\sigma_{k-1}(\lambda_{k-1,j})) + w_k$$

  Loop: $j = J_{min}$ to J_{max}, where J_{min} and J_{max} denote the smallest and the largest integers of j satisfying $\lambda'_{k,1} < \lambda_{k-1,j} + c_k \leq d_k$.

$$i = j - J_{min} + 2$$

$$\lambda'_{k,i} = \lambda_{k-1,j} + c_k$$

$$\sigma'_k(\lambda'_{k,i}) = \sigma_{k-1}(\lambda_{k-1,j}) \oplus \tau_k$$

$$W(\sigma'_k(\lambda'_{k,i})) = W(\sigma_{k-1}(\lambda_{k-1,j})) + w_k$$

 construct σ_k from σ_{k-1} and σ'_k by

  1) merging and sorting λ_{k-1} and λ'_k into one array

  2) making the weights of the schedules in the resultant array strictly increasing; removing any schedule off the array if necessary.

 case 3 $(U_{k-1} < \tau_k)$: (adding one more scheduling point)

$$v_k = v_{k-1} + 1$$

$$\lambda_{k,v_k} = \tau_k + c_k$$

$$\sigma_k(\lambda_{k,v_k}) = \sigma_{k-1}(\lambda_{k-1,v_{k-1}}) \oplus \tau_k$$

79

$$W'(\sigma_k(\lambda_{k,v_k})) = W'(\sigma_{k-1}(\lambda_{k-1,v_{k-1}})) + w_k$$

/* Note that $\lambda_{k,j} = \lambda_{k-1,j}$ and $\sigma_k(\lambda_{k,j}) = \sigma_{k-1}(\lambda_{k-1,j})$ for $j = 1$ to v_{k-1} */

endfor

4   Scheduling a Task Set

To find an optimal schedule for the task set, we may have to consider all possible ($n!$) permutations. It is possible to reduce the search space by eliminating some infeasible permutations. For example, if $d_i < r_j$, there is no feasible schedule in which τ_i is placed after τ_j. Even after the reduction, the search space might still be too large. We propose to use *simulated annealing* technique, recognizing that while this technique reduces the search, it may yield sub-optimal results.

4.1   Simulated Annealing

Simulated annealing is a stochastic approach for solving large optimization problems. It was developed using statistical mechanics ideas to find a global minimum point in the energy space. Kirkpatrick *et al* [5] had demonstrated the power and applications of simulated annealing to the field of combinatorial optimization.

To find the optimal solution of the optimization problem is similar to finding the lowest energy state of metal. The metal is melted first. Then it is cooled down slowly until the freezing point is reached. At each temperature, a number of trials are carried out to reach the equilibrium. The temperature has to be controlled not to drop too quick; otherwise, it is possible to be trapped in a local minimum energy configuration. Lower energy generally indicates a better solution. The annealing process starts from a randomly chosen configuration, proceeding to seek potentially promising neighbor configurations. The neighbor configuration is derived by perturbing the current configuration. If the neighbor configuration has a lower energy, the change is always accepted. The distinct feature is that the neighbor configuration with a higher energy can also be accepted with the probability of $e^{(E-E')/T}$, where T is the temperature, and $E - E'$ represents the difference in the energy of current and neighbor configurations. Notice that when the temperature is high, an energy

up jump is more likely than it is when the temperature is low, as it may reach the configuration, although with higher energy, which may lead to a better solution. An *up jump* means a jump from low energy to high energy, and a *down jump* means a jump from high energy to low energy.

4.2   The Set Scheduling Algorithm (SSA)

A permutation is used to represent the configuration. If a permutation is ordered in an Earliest Deadline First (EDF) fashion, we call it an EDF permutation. An EDF permutation may be a good starting permutation for the process of simulated annealing for this problem. If the window of a task is contained in the window of another task, we say that the latter task contains the former task. If there are no containing relations among tasks, the EDF permutation is a permutation of which an optimal schedule of the task set is a subsequence [4]. Thus, an optimal schedule for the task set can be generated by PSA by scheduling the EDF permutation. The energy function can be expressed by a loss function:

$$loss \ = \ \sum \ weight \ of \ rejected \ noncritical \ tasks$$

A schedule is not acceptable if critical tasks are rejected. We may say that the loss of a rejected critical task is infinity. However, this kind of assignment makes it difficult to distinguish between a very bad schedule (e.g., a critical task is rejected) and even a worse schedule (more critical tasks are rejected). In general, the former schedule can be considered as an improvement over the latter one. If the loss incurred by a rejected critical task is assigned infinity, there is no way to tell which is better between the schedule in which one critical task is rejected and that in which three critical tasks are rejected. Hence, we assign a finite amount of loss to rejected critical tasks. The loss of a critical task must be large enough such that the scheduler will not reject a critical task to accommodate a number of non-critical tasks.

The neighbor function may be obtained using one of the following two methods. In the first, simple method, we randomly select one task from those rejected. This task is inserted in a randomly chosen location within a specified distance from its original location, where the *distance* is the

number of tasks between two tasks in a permutation. The distance is used in this approach to control the degree of perturbation.

The reason of rejecting a task is due to the acceptance of other tasks. Given a schedule for a permutation, it is sometimes difficult to identify which task results in the rejection of other tasks, especially when tasks are congested together. However, the task immediately before or after those rejected is likely to play a role. In the second method, we try to identify the task which causes the largest loss of weight. As a simple approach, we attribute the rejection of a task to the task accepted prior to it. Then we choose the task which causes the largest loss of weight and insert it within a specified distance. Due to the robustness of simulated annealing technique, the impact of not necessarily selecting the task which caused the largest loss is minimal. Note that in simulated annealing many parameters are randomized, and the energy function, together with the temperature, control the progress of the annealing process. Tindell et al [9] commented that the great beauty of the simulated annealing lies in that you only need to describe what constitutes a good solution without worrying about how to reach it. According to our experiments, we find that the first method performs better than the second method. However, the process in the first method sometimes falls into a local minimum. The combination of the two methods does perform better than any of the individual one. The Set Scheduling Algorithm (SSA) is presented in Figure 7.

The initial temperature has to be large enough such that virtually all up jumps are allowed in the beginning of the annealing process. According to [9], the way to compute new temperature is that new temperature = α * current temperature, where $0 \leq \alpha \leq 1$. A step denotes an iteration in the inner loop in Figure 7, which is the process of scheduling a permutation and determining whether the permutation would become the current permutation. The thermal equilibrium can be reached if a certain number of down jumps or a certain number of total steps has been observed; and the freezing point, or the stopping condition, can be reached if no further down jump has been observed in a certain number of steps [5, 9].

Algorithm <u>SSA</u>:

Begin

    choose initial temperature T

    choose edf permutation as the starting permutaion, μ

    schedule μ by PSA and compute its energy, E

    loop

        loop

            compute neighbor permutation μ'

            schedule μ' by PSA and compute its energy, E'

            if $E' < E$ then

                making μ' the current permutation: $\mu \leftarrow \mu'$ and $E \leftarrow E'$

            else

              if $e^{\frac{E-E'}{T}} \geq$ random(0,1) then

                  making μ' the current permutation: $\mu \leftarrow \mu'$ and $E \leftarrow E'$

              else

                μ remains as the current permutation

        until thermal equilibrium is reached

        compute new temperature: $T \leftarrow \alpha * T$

    until stopping condition is reached

End

Figure 7: Set Scheduling Algorithm

5 Experiment Result

Experiments are conducted to study the performance of SSA based on:

- scheduling ability $= \dfrac{\text{number of times that the algorithm generates a feasible schedule}}{\text{number of times that there does exist a feasible schedule for the task set}}$

- loss ratio $= \dfrac{\text{loss of the schedule generated by SSA} - \text{loss of an optimal schedule}}{\text{total weight of accepted noncritical tasks of an optimal schedule}}$

- iterations = number of permutations that the simulated annealing algorithm goes through to obtain the sub-optimal schedule

We start with an EDF permutation. To study how good the result would be by using PSA to schedule the EDF permutation, the scheduling ability and loss ratio for the EDF permutation are computed as well. In our experiments, a task set consists of 100 tasks. The number of permutations in such a task set is $100! \approx 9.33 * 10^{157}$. To study how good the output of SSA is compared to an optimal schedule, it is rather impractical to go through such a great number of permutations for a task set to derive the optimal schedule and its minimum loss for comparison. Instead, we choose to make up a task set such that the task set is feasible and the loss of its optimal schedule is 0. Although the SSA algorithm is primarily designed for an overloaded system, we apply SSA to such task sets for measuring the performance. The parameters are shown in Figure 8.

parameters	value	type
window length	mean_Wl = 20.0	truncated normal distribution
computation time	mean_C $= \frac{mean\_Wl}{3}$	truncated normal distribution
load	20%, 40%, 60%, 80%	constants
criticality ratio	25%, 50%, 75%	constants
weight	low_W=1, high W=50	discrete uniform distribution

Figure 8: Parameters of the experiments

The mean of window length, mean_Wl, is set to be 20 time units. The load is the ratio of total computation time to the largest deadline, D, in the task set. Hence, the load indicates the difficulty

84

of scheduling the task set. The mean of computation time, mean_C, is one third of the mean of window length, which allows the windows among tasks to overlap to some extent. How much the windows overlap partially depends on the load. If the load is high, the windows are congested together, and thus the overlapping is high. We expect some containing relations between tasks to occur and thus increase the difficulty for scheduling. Note that, without containing relations, scheduling the task set would be straightforward. The standard deviations of window length and computation time are set to be their means, respectively. Criticality ratio indicates the percentage of the critical tasks in the task set. It is set to be 25%, 50%, and 75%. The higher the criticality ratio, the more difficult it is to generate a feasible schedule for the task set. On the other hand, although it is easier to come up with a feasible schedule when the criticality ratio is low, the loss ratio may still be high. It may be necessary to go through many permutations before an acceptable loss ratio is reached. In our experiments, the acceptable loss ratio is set to be 0%, which means that SSA will keep trying different permutations until either the loss ratio is 0 or the stopping condition is reached, in which SSA fails to find an optimal schedule. Note that a big energy (loss), 1000, is incurred for a rejected critical task. Hence, for an infeasible schedule, the loss ratio may well be more than 100%. The weight of a non-critical task is an integer ranging from low_W=1 to high_W=50, determined by a discrete uniform distribution function. For each individual experiment with different parameters, 200 task sets, each with 100 tasks, are generated for scheduling. The way of creating a feasible task set without loss is described in appendix A.

From Figure 9a, The scheduling ability of SSA is 98.5% when criticality ratio is 75% and load is 80%, and is 100% for other lower criticality ratios and loads. This is because the simulated annealing algorithm focuses on searching suitable neighbor permutations in such a way that the rejected critical tasks, if any, may be accepted. Note that scheduling only the EDF permutation can not always generate a feasible schedule. The scheduling ability of scheduling EDF permutation degrades when load increases, which means tasks congest more together. The scheduling ability of scheduling EDF permutation also degrades when the criticality ratio increases, which makes meeting the deadlines of all critical tasks become more difficult.

As far as non-critical tasks are concerned, SSA can not guarantee the minimum loss. However, even in the worst case given in Figure 9b, the loss ratio is less than 10%. The loss ratio becomes less when criticality ratio or load is less. In many cases, the loss ratios are less than 5%. As for scheduling the EDF permutation, the loss ratios are significantly larger.

The number of permutations to be searched in simulated annealing depends on the situations of energy jumps, the way of reducing temperature, and how we define thermal equilibrium and stopping conditions. In the experiments, we find that reducing temperature faster does not impose a negative impact on the scheduling ability and loss. How to set the parameters in simulated annealing differs a great deal from one application to another. We do want to generate the result as good as possible, but are not willing to spend more computation time than necessary. This usually requires fine tuning the parameters to get the trade-off between the two goals. We find that the following parameters are beneficial: initial temperature = 3000, $\alpha = 0.8$ (instead of 0.95 or even 0.99 suggested in other applications), the number of down jumps to obtain thermal equilibrium = 25, the number of total steps to obtain thermal equilibrium = 300, the number of steps with no further down jump to obtain the freezing point = 2000, which is also the stopping condition. The average number of permutations searched in simulates annealing is given in Figure 9c. If SSA can successfully generate a feasible schedule, the average number of permutations checked is no more than 4000 times. The number increases a little if SSA fails to find a feasible schedule, because in this case SSA does not stop until the freezing point is reached. Note that the average numbers of permutations are less than n^2, which can roughly give us the idea about the complexity of searching over the permutation space. Additional studies have shown that if we modify the above parameters to increase the average number of permutations by about 10 times, the loss ratios can be further reduced by about 25% of the loss ratios obtained here.

If time can be expressed in integers, the dynamic programming technique used in PSA can be applied by computing $\sigma_k(t)$ at $t = 1, \ldots, D$. Let us call this approach the integral PSA, compared to the original PSA with scheduling points, denoted by PSA SP in Figures 9d. Obviously, the integral PSA tends to compute more schedules than the original PSA. We would like to see how more efficient the original PSA algorithm is than the integral PSA. Specifically, we compare the average
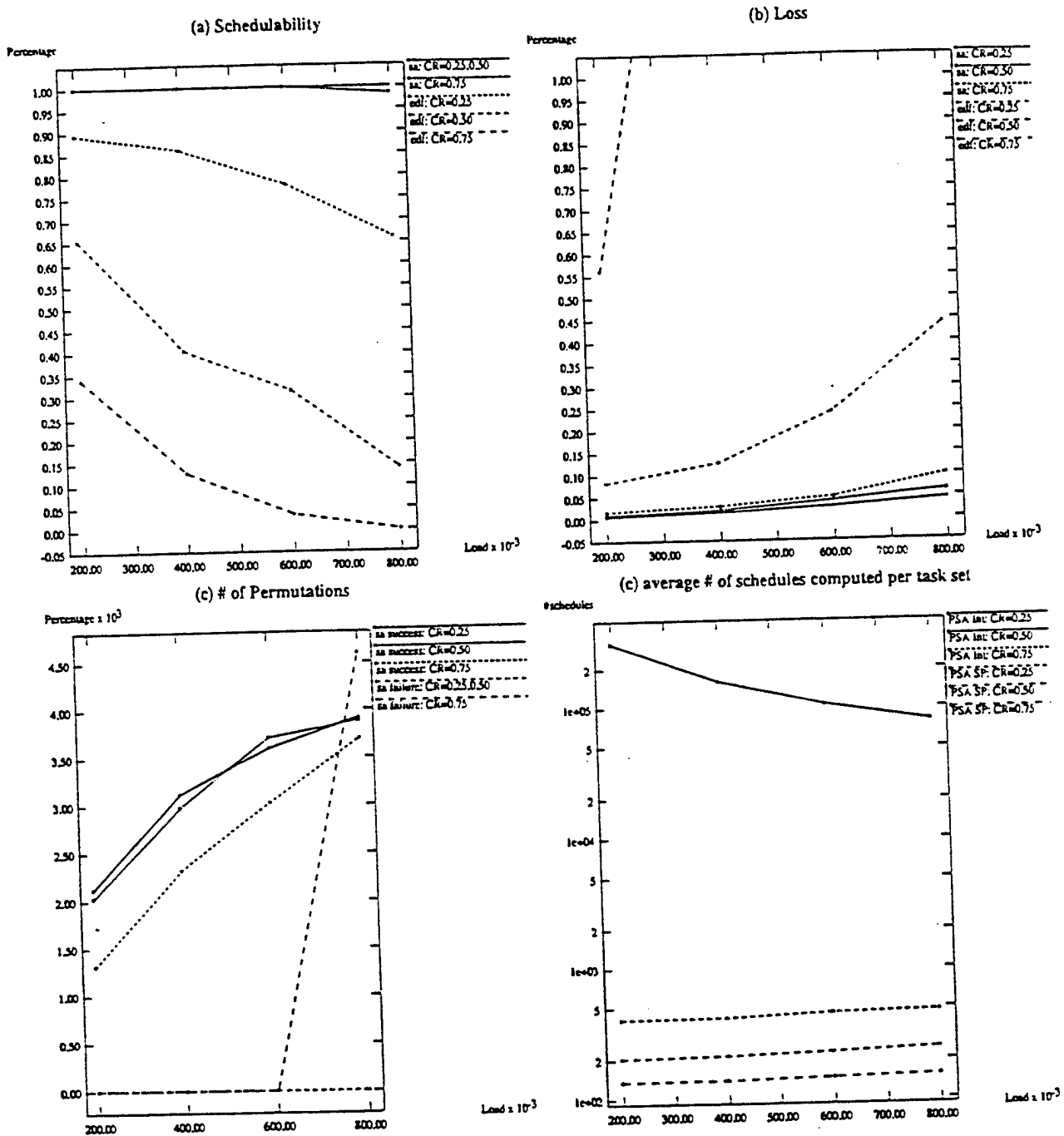
Figure 9: (a) Schedulability; (b) Loss; (c) #Permutations; (d) #Schedules

number of schedules required to derive the optimal schedule for a permutation. For the integral PSA, the number of schedules computed is fixed, or $n * D$, as can be seen in Figure 1. For the original PSA, $\sum_{k=1}^{n} v_k$ is the number of schedules needed to schedule a permutation. The average number of schedules needed to schedule a permutation by PSA is computed over the permutations of a task set, and is presented in Figure 9d. The number for the original PSA decreases with the criticality ratio. This is because a critical task never increases the number of scheduling points; instead, the number of scheduling points might be decreased due to the timing constraint of the critical task. For the criticality ratios of $0.25, 0.50$, and 0.75, the average number of schedules required for a task set of 100 tasks are approximately $480, 250$, and 150, respectively. The complexity of the original PSA seems linear in this sense. On the other hand, the complexity of the integral PSA is quite high. The number decreases with load. This happens to be related to the way of generating the task set, in which $D = total\_c$ / load. The number is equal to $n * D$, where D might fluctuate a little.

6   Conclusion

In this paper, we study the scheduling problem for a real-time system which is overloaded. A significant performance degradation may be observed in the system if the overload problem is not addressed properly [2]. As not all the tasks can be processed, the set of tasks selected for processing is crucial for the proper operation of an overloaded system. We assign to the tasks *criticalities and weights* on the basis of which the tasks are selected. The objective is to generate an optimal schedule for the task set such that all of the critical tasks are accepted, and then the loss of weights of non-critical tasks is minimum.

We present a two step process for generating a schedule. First, we develop a schedule for a permutation of tasks using a pseudo-polynomial algorithm. The concept of *scheduling points* is proposed for the algorithm. In order to find the optimal schedule for the task set, we have to consider all permutations. The simulated annealing technique is used to limit the search space while obtaining optimal or near optimal results. Our experimental results indicate that the approach is

very efficient.

The work presented in this paper can be easily extended to address the overload issue for periodic tasks. To schedule a set of periodic tasks with criticalities and weights, we can convert the periodic tasks in the time frame of the least common multiple of the task periods to aperiodic tasks. The schedule generated for the frame can be applied repeatedly for the subsequent time frames.

Our algorithm can also be applied to solving the problem of scheduling imprecise computations [7], in which a task is decomposed logically into a mandatory subtask, which must finish before the deadline, and an optional subtask, which may not finish. The goal is to find a schedule such that the mandatory subtasks can all be finished by their deadlines and the sum of the computation times of the unfinished optional subtasks is minimum. A schedule satisfies the *0/1 constraint* if every optional subtask is either completed or discarded [7]. We can solve this problem by using our algorithm by setting the mandatory subtasks to be critical, and the optional subtasks to be non-critical with weights equal to their computation times.

Appendix A. Generating a task set

Generate computation times for tasks according to mean_C and the standard deviation
D = (total computation time) / load
Assigning starting instants, s_k, to tasks such that
   the intervals between the computation times are truncated normally distributed
For each task τ_k
   Determine the criticality by criticality_ratio and/or weight by low_W and high_W
   Compute the window length of τ_k according to mean_Wl and the standard deviation
      (note that window length $\geq c_k$)
   align the window with the computation time in their middle points:
      $$r_k = max(0, s_k + \tfrac{c_k}{2} - \tfrac{\text{window\_length}}{2})$$
      $$d_k = min(D, r_k + \text{window\_length})$$

89

The load determines how the tasks would be congested. Once the largest deadline, D, has been computed, we separate the computation times of the tasks in such a way that the *positions* of the computation times on the time axis stretches over the range from 0 to D. Note that the starting instants of the computation times consist in an optimal schedule for the task set. In this way, all of the tasks in the task set can be accepted. At last, the windows are aligned with the computation times.

References

[1] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman Company, San Francisco, 1979.

[2] Jayant R. Haritsa, Miron Livny, and Michael J. Carey. Earliest deadline scheduling for real-time database systems. In *IEEE Real-Time Systems Symposium*, Dec. 1991.

[3] Shyh-In Hwang, Sheng-Tzong Cheng, and Ashok K. Agrawala. An optimal solution for scheduling real-time tasks with rejection. In *International Computer Symposium*, Dec. 1994.

[4] Shyh-In Hwang, Sheng-Tzong Cheng, and Ashok K. Agrawala. Optimization in non-preemptive scheduling for aperiodic tasks. Technical Report CS-TR-3216, UMIACS-TR-94-14, Department of Computer Science, University of Maryland at College Park, Jan. 1994.

[5] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science(220)*, pages 671–680, 1983.

[6] C. L. Liu and J. Layland. Scheduling algorithm for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, Jan. 1973.

[7] W.K. Shih, J. Liu, and J.Y. Chung. Fast algorithms for scheduling imprecise computations. In *IEEE Real-Time Systems Symposium*, pages 12–19, Dec. 1989.

[8] Philip Thambidurai and Kishor S. Trivedi. Transient overloads in fault-tolerant real-time systems. In *IEEE Real-Time Systems Symposium*, Dec. 1989.

[9] K.W. Tindell, A. Burns, and A.J. Wellings. Allocating hard real-time tasks: An np-hard problem made easy. *The Journal of Real-Time Systems*, 4(2):145–165, June 1992.

REPORT DOCUMENTATION PAGE

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	November 1994	Technical Report

4. TITLE AND SUBTITLE

Scheduling an Overloaded Real-Time System

5. FUNDING NUMBERS

N00014-91-C-0195

DASG-60-92-C-0055

6. AUTHOR(S)

Shyh-In Hwang, Chia-Mei Chen and Ashok K. Agrawala

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Maryland
Department of Computer Science
A. V. Willliams Building
College Park, MD 20742

8. PERFORMING ORGANIZATION REPORT NUMBER

CS-TR-3377
UMIACS-TR-94-128

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Honeywell, Inc.
3600 Technology Drive
Minneapolis, MN 55418

Phillips Laboratory
Directorate of Contracting
3651 Lowry Avenue SE
Kirtland AFB NM 87117-5777

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The real-time systems differ from the conventional systems in that every task in the real-time system has a timing constraint. Failure to execute the tasks under the timing constraints may result in fatal errors. Sometimes, it may be impossible to execute all the tasks in the task set under their timing constraints. Considering a system with limited resources, one solution to handle the overload problem is to reject some of the tasks in order to generate a feasible schedule for the rest. In this paper, we consider the problem of scheduling a set of tasks without preemption in which each task is assigned criticality and weight. The goal is to generate an optimal schedule such that all of the critical tasks are scheduled and then the non-critical tasks are included so that the weight of rejected non-critical tasks is minimized. We consider the problem of finding the optimal schedule in two steps. First, we select a permutation sequence of the task set. Secondly, a pseudo-polynomial algorithm is proposed to generate an optimal schedule for the permutation sequence. If the global optimal is desired, all permutation sequences have to be considered. Instead, we propose to incorporate the simulated annealing technique to deal with the large search space. Our experimental results show that our algorithm is able to generate near optimal schedules for the task sets in most cases while considering only a limited number of permutations.

14. SUBJECT TERMS		15. NUMBER OF PAGES
Process Management; Nonnumerical Algorithms and Problems		29
		16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited

Notes on Symbol Dynamics[*][†]

Ashok K. Agrawala

Department of Computer Science, University of Maryland
College Park, Maryland 20742
E-mail: agrawala@cs.umd.edu

Christopher Landauer

System Planning and Development Division, The Aerospace Corporation
The Hallmark Building, Suite 187, 13873 Park Center Road, Herndon, Virginia 22071
Phone: (703) 318-1666, FAX: (703) 318-5409
E-mail: cal@aero.org

13 February 1995

Abstract

This paper introduces a new formulation of dynamic systems that subsumes both the classical discrete and differential equation models as well as current trends in hybrid models. The key idea is to express the system dynamics using symbols to which the notion of time is explicitly attached. The state of the system is described using symbols which are active for a defined period of time. The system dynamics is then represented as relations between the symbolic representations.

We describe the notation and give several examples of its use.

Contents

1 Introduction

Traditionally, systems have been modelled using state variables defined in a metric space and the system dynamics defined using differential equations. This approach uses continuous descriptions of space and time. When we use computers for expressing and manipulating such models we have to use symbols to represent it. Symbols are discrete by their very nature, and require use of mapping from the continuous spaces to discrete spaces. These mappings cause problems unless carried out rather carefully. Further, when we consider the problems in which some aspects of the system are genuinely discrete, hybrid models have been used. As different techniques have to be used for continuous and discrete aspects of the system, significant complexity gets added to such models.

Recognizing that the computer systems only use symbols for any representations, in this paper we present a formulation of system dynamics directly in terms of symbols. In order to handle the synamics, time interval over which a symbol is considered valid is explicitly attached. The symbols describing different aspects of the system may be from a set appropriate for that aspect. The dynamics is described in terms of rules connecting the symbolic representations.

This paper contains the preliminary formulation of system dynamics in the framework of *Symbol Dynamics*.

2 Descriptions of System Behavior

For the purposes of this paper, *behavior* includes all the relationships among parts of a system at the same or different times. In particular, the combined relationships among parts of a system at the same time is usually called *structure*. Both of these aspects are subsumed in our use of the term behavior.

We assume that our ability to generate or derive new information about the system behavior changes only at discrete points in time, since we expect to perform these processes on digital computers. The event times define the time scale. In this paper, we introduce *Symbol Dynamics*, a totally symbolic way to represent the important aspects of dynamical systems and processes, so that we can reason about them using computers.

3 Concepts and Notations

This section contains the basic notions of Symbol Dynamics.

3.1 State Variable

We assume that systems exist and change over time. We are looking for a method of describing those changes so we can compute how to control them.

The systems we consider can be described with state variables. Each state variable is an observation on the system or a derivation from other state variables.

We may or may not know a priori which state variables are important, or even which ones are determinable (i.e., the system comes first, and the state variables are chosen to be helpful in describing the behavior). We might call the state variables *attributes* of the state.

3.2 Symbol

We want to measure and compute with information about a system, so we need to map the system into formal spaces we understand better.

A *type* is a symbol set, both representing a set of values and including some operations on those values; this is the notion of formal space used here. It includes collections of mutually dependent types and functions between different types.

A *symbol* of a given type is an element of the set of values that type. Any notions of credibility, confidence, or uncertainty are part of the type system that is used. It is especially important to define the allowable operations on these kinds of types. For example, for measurements of a system, the symbol would include the measured value and the associated uncertainty value.

3.3 Attribute Identifier

We assume that we will want to know different things about the system behavior. We need names to keep track of the different things we measure or compute.

An *attribute identifier* is a name for a state variable (a state variable is like a probe into some aspect of the system behavior, and the attribute identifier is only the label).

3.4 Expression

An *expression* is a pair
>   (attribute identifier: symbol),

which is interpreted to mean the assertion that the state variable can be described by the symbol (when the expression is active). We will describe the precise semantics of these expressions later on.
These are models of the state variable values.

3.5 Interval

An *interval* is a pair
> |start time, end time),

assumed to describe a half-open interval (to save us from trouble with the topology). The end time may be omitted, in which case it is interpreted to mean infinity by default.

3.6 Characterizer

A *characterizer* is a pair
>   (expression, interval),

also written
>   (attribute identifier: symbol; start time, end time),

interpreted to mean that the expression is *active* during the specified interval. It becomes active at the start time, and becomes inactive at the end time. Each characterizer has a *range* (its *interval of activity*) and a *scope* (the set of attribute identifiers that occur in its expression).
We may also consider a symbol set that includes arithmetic expressions that contain an explicit time variable t. For example,
> $(p : p_0 + v_0 * t; t_0, t_1)$

represents a continuous change along the interval.
We will also have occasion to reason about conditions at particular points in time, so the assertion language will also have characterizers of the form
>   (expression, point).

3.7 Event

An *event* is the activation or deactivation of a characterizer. We make no limiting assumptions about simultaneous events.

4 System Description

A *system description* is a finite set of characterizers, so we assume explicitly that a system can be described by a finite set of characterizers. We insist that only a finite set of characterizers be active at any one time. Since each of those characterizers is active over a positive interval, there is therefore some small interval thereafter during which all of them are still active.
Everything we know about a system's behavior is described by characterizers and relationships among the characterizers. Domain models and context can be written as characterizers, generally with large intervals.

4.1 Dynamics

Relationships among characterizers are rules that define the dynamics. These rules take the form:
>   if *these* characterizers (with a list) are active on *these* intervals, then *this* new one is also active on *this* other interval (not necessarily contained in the intersection of the original intervals).

Rules can contain variable identifiers, with implicit universal quantification.

Relationships hold on intervals and the combination may extend the range. We generate new characterizers according to the relationships, either predictive (range extension) or deductive (knowledge extension).

The language in which the rules are written is important, since it has to accommodate notations from many different types, many of which will not be known when the language is defined. Some basic concepts that will be in any of these languages are continuity and derivatives.

It is important to remember that the system comes first, and that the state variables are our choices for modeling and understanding the system. This means in particular that the coordinate systems we use are temporary, and that the constraints among the state variables are expressed explicitly as relationships.

4.2 Normalization and Continuation

Characterizers may have overlapping intervals. *Normalization* is the process of breaking each characterizer into two or more others, to fit the time scale. If t is an event time, and

$$(a : v; s, e)$$

is a characterizer with $s < t \leq e$, then we can replace it with two characterizers

$$(a : v; s, t) \text{ and } (a : v; t, e).$$

If two characterizers use the same attribute,

$$(a : v; s, e)$$

and

$$(a : w; t, u),$$

then we say that the second one *continues* the first one iff they are adjacent in time, so $t = e$. Continuity considerations in the transition from v to w at time t are treated in the next section.

In any system with a finite density of event times, if we split every characterizer that spans an event time, then we end up with characterizers that start and stop at consecutive event times (though they may be *continued* by other characterizers). This has some computational conveniences.

If we have two characterizers

$$(a : v; t_1, t_2)$$

and

$$(a : w; t_2, t_3),$$

so that the second one continues the first, then we need some kind of explicit characterizer for the transition, active in an interval containing the transition time. If there is a description u in an appropriate domain for which

$$u = \begin{cases} v, & \text{for } t_1 \leq t < t_2, \\ w, & \text{for } t_2 \leq t < t_3, \end{cases}$$

then we can conclude

$$(a : u; t_1, t_3).$$

This is the opposite of normalization.

If there is an overlap, that is, if the two characterizers

$$(a : v; t_1, t_2)$$

and

$$(a : w; t_3, t_4)$$

have

$$[t_1, t_2) \cap [t_3, t_4) \text{ non-empty},$$

and

$$v(t) = w(t) \text{ for } t \in [\max(t_1, t_3), \min(t_2, t_4)),$$

then we can also conclude

$$(a : u; \min(t_1, t_2), \max(t_3, t_4)).$$

4.3 Continuation and Continuity

One aspect of continuity is transitions from one symbol to another across interval boundaries. The transition relations are extra conditions that have to hold *at* the transition time (usually they are smoothness conditions for model transitions).

A typical smoothness property is infinitesimal: for characterizers

$$(a : v; t_0, t_1)$$

and
$$(a : w; t_1, t_2),$$
we normally want smoothness, written
$$\left. \frac{d\,v}{d\,t} \right|_{t=t_1^-} = \left. \frac{d\,w}{d\,t} \right|_{t=t_1^+},$$
and continuity, written
$$v(t = t_1^-) = w(t = t_1^+).$$
Both of these are point conditions on the attributes and their derivatives, and we can consider only conditions on attributes by using whatever derivatives are needed in the conditions: instead of
$$(a : v; t_0, t_1),$$
we use
$$(a : (v, v'); t_0, t_1),$$
and write our smoothness condition as
$$\left(\begin{array}{c} v \\ v' \end{array} \right)_{t=t_0^-} = \left(\begin{array}{c} w \\ w' \end{array} \right)_{t=t_0^+}.$$
If we also require continuity in each attribute, so that
$$w(t = t_1^+) = w(t = t_1),$$
then the upper limit in the previous expression can be omitted.

It is therefore clear that we must deal with point events at transitions
$$[t_0...t_1) \, [t_1...t_2),$$
but not with point characterizers. If we make the transition continuity a property of the definition of continuation, then we can assert it or not in any given model.

Of course, the expression $t = t_1^-$ means that the interval $[t_1 - \epsilon, t_1)$ is part of the limit computation for every ϵ small enough, so we might be able to use these intervals for some small enough ϵ without having to take the limits.

We will deal with these considerations in the simplest way possible. We have a characterizer that asserts continuity of the relevant attribute across a larger interval, such as $[t_0, t_2)$ above. The only place that the continuity characterizer has new information is at the transition point t_1, but we simply do not worry about the redundancy.

4.4 Characterizer Semantics and Inference

A characterizer is what we want to assume about what is true over its interval. It need not be consistent with the other characterizers in a system description; we explicitly allow false assertions here, so we can reason using counterfactuals.

4.4.1 Inference

We can make inferences within intervals, according to some rules. If, say, there is a rule
$$s_1 \& s_2 \implies s_3,$$
and two characterizers
$$(v : s_1; t_0, t_1)$$
and
$$(v : s_2; t_2, t_3)$$
with $t_0 < t_2 < t_1 < t_3$, then we can conclude
$$(v : s_3; t_2, t_1).$$

4.4.2 Prediction

We can also make inferences that extend intervals in some cases. They take the form: If
$$(v : s_1; t_0, t_1)$$
and
$$(w : s_2; t_0, t_1)$$
are characterizers with $t_0 < t_1$, then there is a characterizer
$$(z : s_3; t_2, t_3)$$
for some t_2, t_3, with $t_0 < t_2 < t_1 < t_3$.

4.4.3 Truth Maintenance

Because we do not presume that the characterizers in a system are truths, we need to be much more careful about when they can be used together, especially in the inference and prediction processes. Since the inference rules themselves are time dependent, we need to keep track of the dependencies of every characterizer, both how and when it was derived (how tells us about hypotheses and inference rules; when helps us in checking temporal consistency) and its interval of activity.

We also need a way to indicate which characterizers we DO want to be true, so that different collections of characterizers can be compared and contrasted within the same context. We might want to consider computing various maximal consistent sets of irredundant assertions as an aid in this process.

Various rules can be activated that lead to new conclusions in an interval, which can supersede old ones; we also assume partial deduction, not total. We therefore need to use some kind of non-monotonic logic.

4.5 Analysis

Simulation is a continuing surprise.

We want tools with analytic power to help reduce our reliance on simulation, so we can make reliable predictions about the system behavior.

All of our computations are performed from the symbols active at a given time. The advantage of dealing explicitly with time in this formulation is that we can sit outside the usual sequencing of events, taking a kind of "side-long" look at the entire time line, and piece together parts of the models that we know more about regardless of whether or not they are the first ones in our time interval of interest.

We can also perform the deductions in an order that is different from the order imposed by time, using any of a number of simple mechanisms, such as rule-based systems or rewrite logics; both are being investigated.

5 Examples

This section contains several examples that illustrate the utility of the notation.

5.1 ODE

A simple example that shows range extension is an ordinary differential equation (ODE). For ODEs, the solution method is part of changing an ODE into a set of characterizers.

So let us consider a simple second-order ODE for the sine function,

$$y'' = -y,$$
$$y'(0) = 1,$$
$$y(0) = 0,$$

and solve it with Euler's method (a particularly bad one for this kind of problem, by the way).

First, we transform the equations into a first order system (in the usual way) by taking $x = y'$,

$$x' = -y,$$
$$y' = x,$$
$$x(0) = 1,$$
$$y(0) = 0,$$

and we also define $z = x' = y''$.

5.1.1 First-Order

Now the way Euler's method works is by linear extrapolation, so for a given time $t = t_0$, if we have

$$x(t_0) = x_0,$$
$$y(t_0) = y_0,$$

then we have

$$z_0 = z(t_0) = -y_0,$$

and we take

$$x(t) = x_0 + z_0 * (t - t_0),$$
$$y(t) = y_0 + x_0 * (t - t_0),$$

for t in some small interval

$[t_0, t_1 = t_0 + dt)$.

The characterizers that describe this situation are:

$(x : x_0 + z_0 * (t - t_0); t_0, t_0 + dt)$,

$(y : y_0 + x_0 * (t - t_0); t_0, t_0 + dt)$,

which we want to be true for all choices of x_0, y_0, t_0, and dt (which ones we actually use in our system description depend on how we choose the time intervals in the solution).

The characterizers that describe the initial conditions are difficult, because they cannot be described with half-open intervals of the shape we have thus far described:

$(x : 1; 0)$,

$(y : 0; 0)$,

which is always going to be a problem in systems that start at a certain time.

In a more sophisticated system, the choice of next time interval would depend on the computed accuracy of the current solution.

For this example, we simply make all the time intervals the same, and say that the characterizer pair

$(x : x_1 + z_1 * (t - t_1); t_1, t_1 + dt)$,

$(y : y_1 + x_1 * (t - t_1); t_1, t_1 + dt)$

propagates the pair

$(x : x_0 + z_0 * (t - t_0); t_0, t_0 + dt)$,

$(y : y_0 + x_0 * (t - t_0); t_0, t_0 + dt)$

iff

$$x_1 = x_0 + z_0 * dt,$$
$$y_1 = y_0 + x_0 * dt,$$
$$t_1 = t_0 + dt,$$

which are the conditions for the first pair to meet the second (the condition $z_1 = -y_1$ is part of the definition of these characterizer pairs).

Extending the iteration, we have

$$x(0) = 1,$$
$$y(0) = 0,$$
$$x(k+1) = x(k) - y(k) * dt,$$
$$y(k+1) = y(k) + x(k) * dt,$$

which can be written as a vector equation (we put the matrix on the right so we can use row vectors)

$$(x, y)(0) = (1, 0),$$

$$(x, y)(k+1) = (x, y)(k) \begin{pmatrix} 1 & dt \\ -dt & 1 \end{pmatrix},$$

so if we write I for the identity matrix and J for the matrix

$$\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix},$$

then we have (with $X = (x, y)$)

$$X(0) = (1, 0),$$
$$X(k+1) = X(k)(I + J * dt),$$

so

$$X(k) = (1, 0)(I + J * dt)^k,$$

which can be computed exactly.

Since the eigenvalues of $(I + J * dt)$ are $1 \pm i * dt$, which have magnitude $1 + dt^2$, the successive powers of the matrix diverge for any $dt > 0$, and therefore so does the iteration.

5.1.2 Second-Order Example

In this section, we use the same differential equation problem, with a different solver, a second-order one that is almost able to converge properly. We therefore have

$$x' = -y,$$

100

$$y' = x,$$
$$x(0) = 1,$$
$$y(0) = 0,$$

as above. Our initial conditions are

$$(x : 1; 0),$$
$$(y : 0; 0),$$

as before.

The method we use is a simplified second-order Runge-Kutta method [?], [?], which basically amounts to averaging the usual Euler approximation in an interval with a linear reapproximation at the endpoint of the interval. At a given time $t = t_0$, if we have

$$x(t_0) = x_0,$$
$$y(t_0) = y_0,$$

then we have

$$x(t) = x_0 - y_0 * dt - x_0 * dt^2/2,$$
$$y(t) = y_0 + x_0 * dt - y_0 * dt^2/2,$$

and it is the extra dt^2 terms that make the method second-order.

As above, we assume equal time intervals and get an iteration

$$x(0) = 1,$$
$$y(0) = 0,$$
$$x(k+1) = x(k) - y(k) * dt - x(k) * dt^2/2,$$
$$y(k+1) = y(k) + x(k) * dt - y(k) * dt^2/2,$$

which can be written as a vector equation

$$(x, y)(0) = (1, 0),$$

$$(x, y)(k+1) = (x, y)(k) \begin{pmatrix} 1 - dt^2/2 & dt \\ -dt & 1 - dt^2/2 \end{pmatrix},$$

and we have as above

$$X(0) = (1, 0),$$
$$X(k+1) = X(k)(I * (1 - dt^2/2) + J * dt),$$

so

$$X(k) = (1, 0) (I * (1 - dt^2/2) + J * dt)^k,$$

which can be computed exactly.

Since the eigenvalues of $(I * (1 - dt^2/2) + J * dt)$ are $1 - dt^2/2 \pm i * dt$, which have magnitude $1 + dt^4/4$, this simple method still does not converge (but much more slowly).

5.1.3   Higher-Order Example

A similar analysis of the usual 4th-order Runge-Kutta method leads to an iteration

$$x(t) = x_0 - y_0 * dt - x_0 * dt^2/2 + y_0 * dt^3/6 + x_0 * dt^4/24,$$
$$y(t) = y_0 + x_0 * dt - y_0 * dt^2/2 - x_0 * dt^3/6 + y_0 * dt^4/24,$$

with matrix

$$\begin{pmatrix} 1 - dt^2/2 + dt^4/24 & dt - dt^3/6 \\ -dt + dt^3/6 & 1 - dt^2/2 + dt^4/24 \end{pmatrix},$$

and eigenvalue magnitude of $1 + dt^6/36 + dt^8/24^2$, which is still greater than one. In fact, since this equation (in (x, y) space) represents moving around a circle, any extrapolation method based on tangents at a single point will fail, since all of the tangent vectors point outward from the circle. We note that the iteration equations do have the first terms of the usual Maclaurin series for $\sin(dt)$ and $\cos(dt)$, so we try out a different iteration:

$$x(t) = x_0 * \cos(dt) - y_0 * \sin(dt),$$
$$y(t) = y_0 * \cos(dt) + x_0 * \sin(dt),$$

which can be written as a vector equation

$$(x, y)(0) = (1, 0),$$

$$(x,y)(k+1) = (x,y)(k) \begin{pmatrix} cos(dt) & sin(dt) \\ -sin(dt) & cos(dt) \end{pmatrix},$$

and we have as above

$$X(0) = (1,0),$$
$$X(k+1) = X(k)(I * cos(dt) + J * sin(dt)),$$

so

$$X(k) = (1,0) (I * cos(dt) + J * sin(dt))^k,$$
$$= (1,0) (I * cos(k * dt) + J * sin(k * dt)),$$

and

$$x(k * dt) = cos(k * dt),$$
$$y(k * dt) = sin(k * dt),$$

from which we can hazard a guess as to the correct solution.

5.2 Measurement

Let us take a simple system in which the velocity and position are occasionally known through inexact measurement. Our state variables are p for the position, v for the velocity, and a for the unknown acceleration. We assume that the acceleration a is bounded by some constant A, so that for any times $t_0 < t_1$

$$|v(t_1) - v(t_0)| < |t_1 - t_0| * A.$$

We assume that we have characterizers

$$(a(t); t_{i-1}, t_i)$$

that describe the acceleration, and model characterizers

$$(v = p'; 0^-, -),$$
$$(a = v'; 0^-, -).$$

Therefore, we can compute the velocity and position by

$$v(t) = v(t_0) + \int_{t_0 < u < t} a(u)\, du,$$

$$p(t) = p(t_0) + \int_{t_0 < u < t} v(u)\, du.$$

The problem is to choose measurement times and variables that maintain a certain accuracy in the estimates of position.

We assume that we can measure position within a bound

$$|p_{meas}(t) - p(t)| < P,$$

and that we can measure velocity within a bound

$$|v_{meas}(t) - v(t)| < V,$$

but that we want to keep our estimate p_{est} of position either more accurately than the position measurement error (this might or might not be possible) or using as few measurements as possible.

We assume first that x_0, v_0 are known, and consider an interval $[t_0, t_1)$. We compute

$$|v(t_1) - v_0| < |t_1 - t_0| * A,$$

and therefore

$$|x(t_1) - x_0| < \frac{1}{2} * |t_1 - t_0|^2 * A,$$

so we would have to choose

$$\Delta t = t_1 - t_0$$

so that

$$\Delta t \leq |V/A|$$

to keep the velocity within bounds, and

$$(\Delta t)^2 \leq |2 * P/A|$$

to keep the position within bounds.

But of course, we don't know $x(t)$ or $v(t)$ after the first time interval, so we need to change the previous derivation a bit.

We assume that we know x_0 and v_0, and that

$$|x(t_0) - x_0| \leq \Delta x_0$$

describes the accuracy of our knowledge of $x(t)$ at time $t = t_0$, and

$$|v(t_0) - v_0| \leq \Delta v_0$$

describes the accuracy of our knowledge of $v(t)$ at time $t = t_0$. Then the above inequalities become

$$|v(t_1) - v_0| \leq \Delta v_0 + |t_1 - t_0| * A,$$

and therefore

$$|x(t_1) - x_0| \leq \Delta x_0 + |t_1 - t_0| * \Delta v_0 + \frac{1}{2} * |t_1 - t_0|^2 * A,$$

so we have to have

$$\Delta t \leq |(V - \Delta v_0)/A|$$

to keep the velocity within bounds, and

$$(\Delta t)^2 + \frac{2 * \Delta v_0}{A} * (\Delta t) \leq |2 * (P - \Delta x_0)/A|$$

to keep the position within bounds.

At this point, we are stuck unless we can say something more helpful about the acceleration. Suppose we know that the acceleration jumps around, and that it has a distribution of values with mean 0 and variance R. In this case, we might be able to reduce the estimates for position and velocity and improve the time intervals.

References

[1] P. Henrici, Elements of Numerical Analysis, Wiley (1964)

[2] J. Stoer, R. Bulirsch, Einfu:hrung in die Numerische Mathematik, II Springer (1973)

REPORT DOCUMENTATION PAGE

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	February 13, 1995	Technical Report

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Notes on Symbol Dynamics	N00014-91-C-0195 and DSAG-60-92-C-0055

6. AUTHOR(S)

Ashok K. Agrawala and Christopher Landauer

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
University of Maryland A.V. Williams Building College Park, Maryland 20742	CS-TR - 3411 UMIACS-TR - 95-15

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING / MONITORING AGENCY REPORT NUMBER
Honeywell 3660 Technology Drive Minneapolis, MN 55418 Phillips Labs 3550 Aberdeen Ave. SE Kirtland AFB, NM 87117-5776	

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This paper introduces a new formulation of dynamic systems that subsumes both the classical discrete and differential equation models as well as current trends in hybrid models. The key idea is to express the system dynamics using symbols to which the notion of time is explicitly attached. The state of the system is described using symbols which are active for a defined period of time. The system dynamics is then represented as relations between the symbolic representations. We describe the notation and give several examples of its use.

14. SUBJECT TERMS			15. NUMBER OF PAGES
C.m, Miscellaneous			11 pages
			16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABS
Unclassified	Unclassified	Unclassified	Unlimited

NSN 7540-01-280-5500

Standard Form 298 (Rev.

Allocation and Scheduling of Real-Time Periodic Tasks with Relative Timing Constraints*

Sheng-Tzong Cheng and Ashok K. Agrawala
Institute for Advanced Computer Studies
Systems Design and Analysis Group
Department of Computer Science
University of Maryland
College Park, MD 20742
{stcheng,agrawala}@cs.umd.edu

Abstract

Allocation problem has always been one of the fundamental issues of building the applications in distributed computing systems (DCS). For real-time applications on DCS, the allocation problem should directly address the issues of task and communication scheduling. In this context, the allocation of tasks has to fully utilize the available processors and the scheduling of tasks has to meet the specified timing constraints. Clearly, the execution of tasks under the allocation and schedule has to satisfy the precedence, resources, and other synchronization constraints among them.

Recently, the timing requirements of the real-time systems emerge that the relative timing constraints are imposed on the consecutive executions of each task and the inter-task temporal relationships are specified across task periods. In this paper we consider the allocation and scheduling problem of the periodic tasks with such timing requirements. Given a set of periodic tasks, we consider the least common multiple (LCM) of the task periods. Each task is extended to several instances within the LCM. The scheduling window for each task instance is derived to satisfy the timing constraints. We develop a simulated annealing algorithm as the overall control algorithm. An example problem of the sanitized version of the Boeing 777 Aircraft Information Management System is solved by the algorithm. Experimental results show that the algorithm solves the problem in a reasonable time complexity.

1 Introduction

The task allocation and scheduling problem is one of the basic issues of building real-time applications on a distributed computing system (DCS). DCS is typically modeled as a collection of processors interconnected by a communication network. For hard real-time applications, the allocation of tasks over DCS is to fully utilize the available processors and the scheduling is to meet their timing constraints. Failure to meet the specified timing constraints or inability to respond correctly can result in disastrous consequence.

For the hard real-time applications, such as avionics systems and nuclear power systems, the approach to guarantee the critical timing constraints is to allocate and schedule tasks *a priori*. The essential solution is to find an static allocation in which there exists a feasible schedule for the given task sets. Ramamritham [Ram90] proposes a global view where the purpose of allocation should directly address the schedulability of processors and communication network. A heuristic approach is taken to determine an allocation and find a feasible schedule under the allocation. Tindell *et al.* [TBW92] take the same global view and exploit a simulated annealing technique to allocate periodic tasks. A distributed rate-monotonic scheduling algorithm is implemented. In each period a task must execute once before the specified deadline. The transmission times for the communications are taken into account by subtracting the total communication time from the deadline and making the execution of the task more stringent.

Simply assuring that one instance of each task starts after the ready time and completes before the specified deadline is not enough. Some real-time applications have more complicated timing constraints for the tasks. For example, the relative timing constraints may be imposed upon the consecutive executions of a task in which the scheduling of two consecutive executions of a periodic task must be separated by a minimum execution interval. Communication latency can be specified to make sure that the time difference between the completion of the sending task and the start of the receiving task does not exceed the specified value. The Boeing 777 Aircraft Information Management System is such an example [CDHC94]. For such applications, the algorithms proposed in literature do not work because the timing constraints are imposed across the periods of tasks. In this paper, we consider the relative timing constraints for real examples of real-time applications in Section 2. Based on the task characteristics, we propose the approach to allocate and schedule these applications in Section 3. A simulated annealing algorithm is developed to solve the problem in which the reduction on the search space is given in Section 4. In Section 5, we evaluate the practicality and show the significance of the algorithm. Instead of randomly generating the *ad hoc* test cases, we apply the algorithm to a real example. The example is the Boeing 777 AIMS with various numbers of processors. The experimental results are shown in Section 5.

2 Problem Description

Various kinds of periodic task models have been proposed to represent the real-time system characteristics. One of them is to model an application as an independent set of tasks, in which each task is executed once every period under the ready time and deadline constraints. Synchronization (e.g. precedence and mutual exclusion) and communications are simply ignored. Another model to take the precedence relationship and communications into account is to model the application as a task graph. In a task graph, tasks are represented as nodes while communications and precedence relationship between tasks are represented as edges. The absolute timing constraints can be imposed on the tasks. Tasks have to be allocated and scheduled to meet their ready time and deadline constraints upon the presence of synchronization and communications. The deficiency of task graph modeling is inability of specifying the relative constraints across task periods. For example, one can not specify the minimum separation interval between two consecutive executions of the same task.

In the work [CA93], we modified the real-time system characteristics by taking into account the relative constraints on the instances of a task. We considered the scheduling problem of the periodic tasks with the relative timing constraints. We analyzed the timing constraints and derive the scheduling window for each task instance. Based on the scheduling window, we presented the time-based approach of scheduling a task instance. The task instances are scheduled one by one based on their priorities assigned by the proposed algorithms. In this paper we augment the real-time system characteristics by considering the inter-task communication on DCS.

2.1 Task Characteristics

The problem considered in this chapter has the following characteristics.

- The Fundamentals: A task is denoted by the 4-tuple $< p_i, e_i, \lambda_i, \eta_i >$ denoting the period, computation time, low jitter and high jitter respectively. One instance of a task is executed each period. The execution of a task instance is non-preemptable. The start times of two consecutive instances of task τ_i are at least $p_i - \lambda_i$ and at most $p_i + \eta_i$ apart. Let s_i^j and f_i^j be the start time and finish time of task instance τ_i^j respectively. The timing constraints specified in Equations 1 through 4 must be satisfied.

$$f_i^j = s_i^j + e_i \tag{1}$$

$$s_i^{n_i+1} = s_i^1 + \text{LCM} \tag{2}$$

$$s_i^j \geq s_i^{j-1} + p_i - \lambda_i \tag{3}$$

$$s_i^j \leq s_i^{j-1} + p_i + \eta_i \tag{4}$$

$$\forall j = 2, \ldots, n_i + 1.$$

- **Asynchronous Communication:** Tasks communicate with each others by sending and receiving data or messages. The frequencies of sending and receiving tasks of a communication can be different. In consequence, communications between tasks may cross the task periods. When such asynchronous communications occur, the semantics of undersampling is assumed. When two tasks of different frequencies are communicating, schedule the message only at the lower rate. For example, if task A (of 10HZ) sends a message to task B (of 5HZ), then in every 200ms, one of two instances of task A has to send a message to one instance of task B. If the sending and receiving tasks are assigned to the same processor, then a local communication occurs. We assume the time taken by a local communication is negligible. When an interprocessor communication (IPC) occurs, the communication must be scheduled on the communications network between the end of the sending task execution and the start of the receiving task execution. The transmission time required to communicate the message i over the network is denoted by μ_i.

- **Communication Latency:** Each communication is associated with a communication latency which specifies the maximum separation between the start time of the sending task and the completion time of the receiving task.

- **Cyclic Dependency:** Research on the allocation problem has usually focused on acyclic task graphs [Ram90, HS92]. Given an acyclic task graph $G = \{V, E\}$, if the edge from task A to task B is in E then the edge from B to A can not be in E. The use of acyclic task graphs excludes the possibility of specifying the cyclic dependency among tasks. For example, consider the following situation in which one instance of task A can not start its execution until it receives data from the last instance of task B. After the instance of task A finished its execution, it sends data to the next instance of task B. Since tasks A and B are periodic, the communication pattern goes on throughout the lifetime of the application. To be able to accommodate this situation, we take cyclic dependency into consideration.

The timing constraints described above are shown in Figure 1. For periodic tasks A and B, the start times of each and every instance of task execution and communication are pre-scheduled such that (1) the execution intervals fall into the range between $p - \lambda$ and $p + \eta$ and (2) the time window between the start time of sending task and the completion time of receiving task is less than the latency of the communication. In Figure 2, we illustrate examples of all possible communication patterns considered in this paper. The description of the communications in the task system is in the form of "*From sender-task-id (of frequency) To receiver-task-id (of frequency)*". If the sender

$$p_A - \lambda_A \le T \le p_A + \eta_A$$

$Proc_1$:   A   A

T < Latency (B to A)

$$p_B - \lambda_B \le T \le p_B + \eta_B$$

T < Latency (A to B)

$Proc_2$:   B   B

A to B   B to A   A to B

Network:
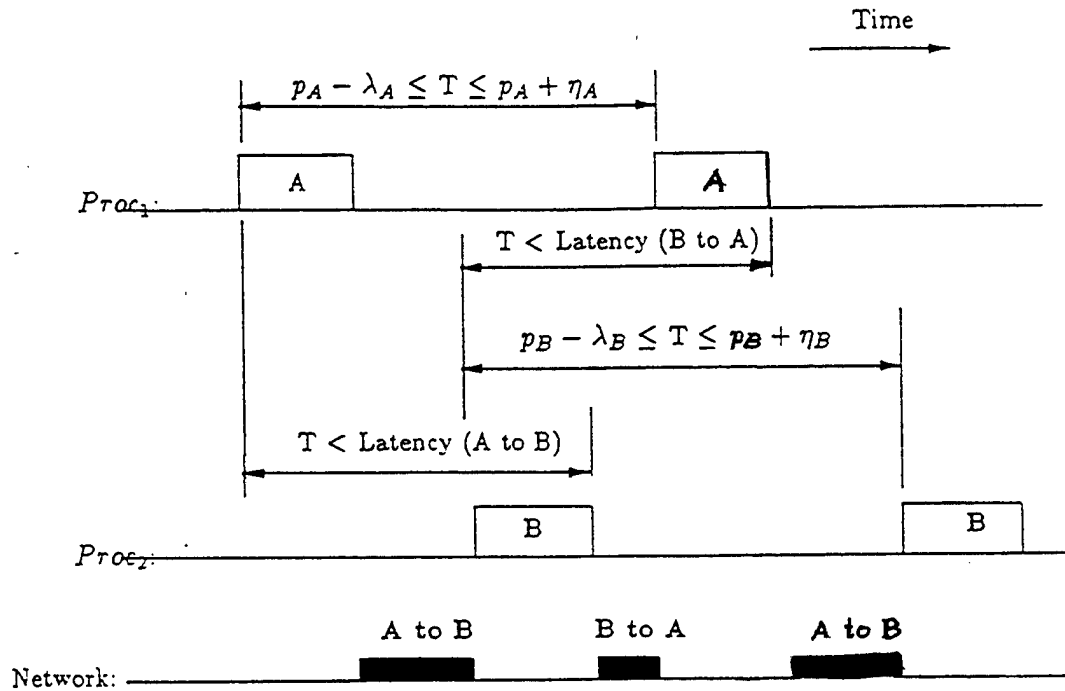
Figure 1: Relative Timing Constraints

frequency is n times of the receiver frequency and no cyclic dependency is involved, then one of every n instances of the sending task has to communicate with one instance of the receiving task. (Examples of this situation are shown in Figures 2.a.1 and 2.a.2. Likewise, for the case in which the receiver frequency is n time that of the sender frequency and no cyclic dependency is present, the patterns are shown in Figures 2.b.1 and 2.b.2. For an asynchronous communication, the sending (receiving) task in low frequency sends (receives) the message to (from) the *nearest* receiving (sending) task as shown in Figure 2.a (2.b). The cases where cyclic dependency is considered are shown in Figures 2.c and 2.d.

2.2 System Model

A real-time DCS consists of a number of processors connected together by a communications network. The execution of an instance on a processor is nonpreemptable. To provide predictable communication and to avoid contention for the communication channel at the run time, we make the following assumptions. (1) Each IPC occurs at the pre-scheduled time as the schedule is generated. (2) At most one communication can occur at any given time on the network.

(a.1)

(a.2)

200 ms
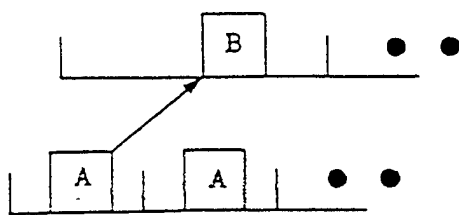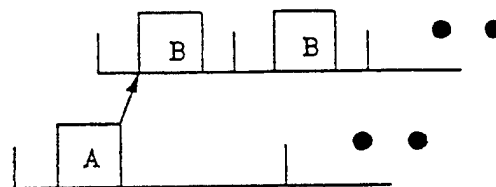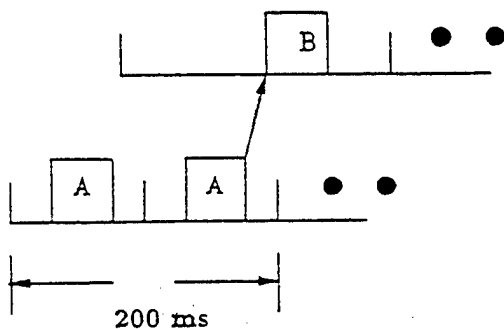
From A (of 10HZ) to B (of 5HZ)

(b.1)

(b.2)

200 ms
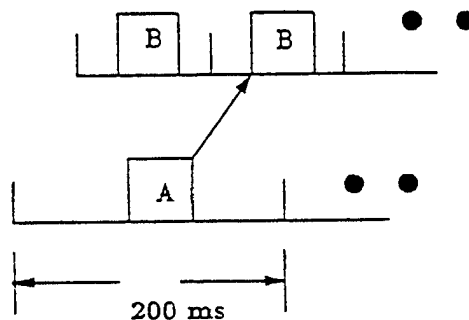
From A (of 5HZ) to B (of 10HZ)

(c)

200 ms
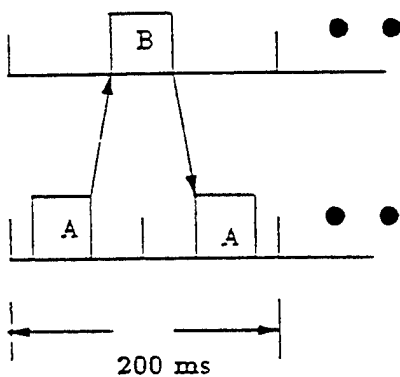
From A (of 10HZ) to B (of 5HZ)

From B (of 5HZ) to A (of 10HZ)

(d)

200 ms
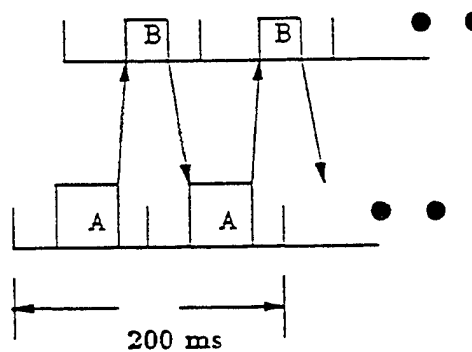
From A (of 10HZ) to B (of 10HZ)

From B (of 10HZ) to A (of 10HZ)

Figure 2: Possible Communication Patterns

110

2.3 Problem Formulation

We consider the static assignment and scheduling in which a task is the finest granularity object of assignment and an instance is the unit of scheduling. We applied the simulated annealing algorithm [KGV83] to solve the problem of real-time periodic task assignment and scheduling with hybrid timing constraints. In order to make the execution of instances satisfy the specifications and meet the timing constraints, we consider a scheduling frame whose length is the least common multiple (LCM) of all periods of tasks. Given a task set Γ and its communications C, we construct a set of task instances, I, and a set of multiple communications, M. We extend each task $\tau_i \in \Gamma$ to n_i instances, $\tau_i^1, \tau_i^2, \ldots,$ and $\tau_i^{n_i}$. These n_i instances are added to I. Each communication $\tau_i \mapsto \tau_j \in C$ is extended to $\min(n_i, n_j)$[1] undersampled communications where $n_i = \text{LCM}/p_i$ and $n_j = \text{LCM}/p_j$. These multiple communications are added to M. The extension can be stated as follows.

- If $n_i < n_j$, then $\tau_i \mapsto \tau_j$ is extended to $\tau_i^1 \mapsto \tau_j^?, \tau_i^2 \mapsto \tau_j^?, \ldots,$ and $\tau_i^{n_i} \mapsto \tau_j^?$.

- If $n_i > n_j$, then $\tau_i \mapsto \tau_j$ is extended to $\tau_i^? \mapsto \tau_j^1, \tau_i^? \mapsto \tau_j^2, \ldots,$ and $\tau_i^? \mapsto \tau_j^{n_j}$.

- If $n_i = n_j$, then $\tau_i \mapsto \tau_j$ is extended to $\tau_i^1 \mapsto \tau_j^1, \tau_i^2 \mapsto \tau_j^2, \ldots,$ and $\tau_i^{n_i} \mapsto \tau_j^{n_j}$.

A task ID with a superscript of question mark indicates some instance of the task. For example, $\tau_i^1 \mapsto \tau_j^?$ means that τ_i^1 communicates with some instance of τ_j. We describe how we assign the nearest instance for each communication in Section 4.1.2.

The problem can be formulated as follows. Given a set of task instance, I, its communications M, we find an assignment ϕ, a total ordering σ_m of all instances, and a total ordering σ_c of all communications to minimize

$$
\begin{aligned}
E(\phi, \sigma_m, \sigma_c) \quad = \quad & \sum_{i,j} \delta(p_i - \lambda_i - s_i^{j+1} + s_i^j) + \sum_{i,j} \delta(s_i^{j+1} - s_i^j - p_i - \eta_i) \\
& + \sum_{i,j} \delta(f_i^j - d_i^j) + \sum_{i,j,k,l} \delta(F(t_i^j \mapsto t_k^l, \sigma_c) - s_k^l) \\
& + \sum_{i,j,k,l} \delta(f_k^l - s_i^j - \text{Latency}\,(\tau_i \text{ to } \tau_k)) \quad\quad (5)
\end{aligned}
$$

subject to $s_i^j \geq r_i^j$ and $S(t_i^j \mapsto t_k^l, \sigma_c) \geq f_i^j, \ \forall t_i^j \mapsto t_k^l,$

where

---

[1]Due to undersampling, when an asynchronous communication is extended to multiple communications, the number of multiple communications is the smaller number of sender and receiver instances.

- s_i^j is the start time of τ_i^j under σ_m.

- f_i^j is the completion time of τ_i^j under σ_m.

- $r_i^j = p_i \times (j-1) + r_i$, and $d_i^j = p_i \times (j-1) + d_i$.

- $\delta(x) = 0$, if $x \leq 0$; and $= x$, if $x > 0$.

- $\phi(\tau_i)$ is the ID of processor which τ_i is assigned to.

- $\tau_i^j \mapsto t_k^l$ is the communication from τ_i^j to τ_k^l. If $\phi(\tau_i) = \phi(\tau_k)$, then $\tau_i^j \mapsto \tau_k^l$ is a local communication.

- $S(c, \sigma_c)$ is the start time of communication c on the network under σ_c.

- $F(c, \sigma_c)$ is the completion time of communication c on the network under σ_c.

The minimum value of $E(\phi, \sigma_m, \sigma_c)$ is zero. It occurs when the executions of all instances meet the jitter constraints and all communications meet their latency constraints. A feasible multiprocessor schedule can be obtained by collecting the values of s_i^j and f_i^j, $\forall$ i and j. Likewise, a feasible network schedule can be obtained from $S(c, \sigma_c)s$ and $F(c, \sigma_c)s$.

Since the task system is asynchronous and the communication pattern could be in the form of cyclic dependency, we solve the problem of finding a feasible solution $(\phi, \sigma_m, \sigma_c)$ by exploiting the cyclic scheduling technique and embedding the technique into the simulated annealing algorithm.

3   The Approach

3.1   Bounds of a Scheduling Window

Define the scheduling window for a task instance as the time interval during which the task can start. Traditionally, the lower and upper bounds of the scheduling window for a task instance are called earliest start time (est) and latest start time (lst) respectively. These values are given and independent of the start times of the preceding instances.

We consider the scheduling of periodic tasks with relative timing constraints described in Equations 3 and 4. The scheduling window for a task instance is derived from the start times of its preceding instances. A *feasible* scheduling window for a task instance τ_i^j is a scheduling window in which any start time in the window makes the timing relation between s_i^{j-1} and s_i^j satisfy

Before: **After:**

LCM | 0 LCM | 0

F F

τ $\tau + e - \text{LCM}$
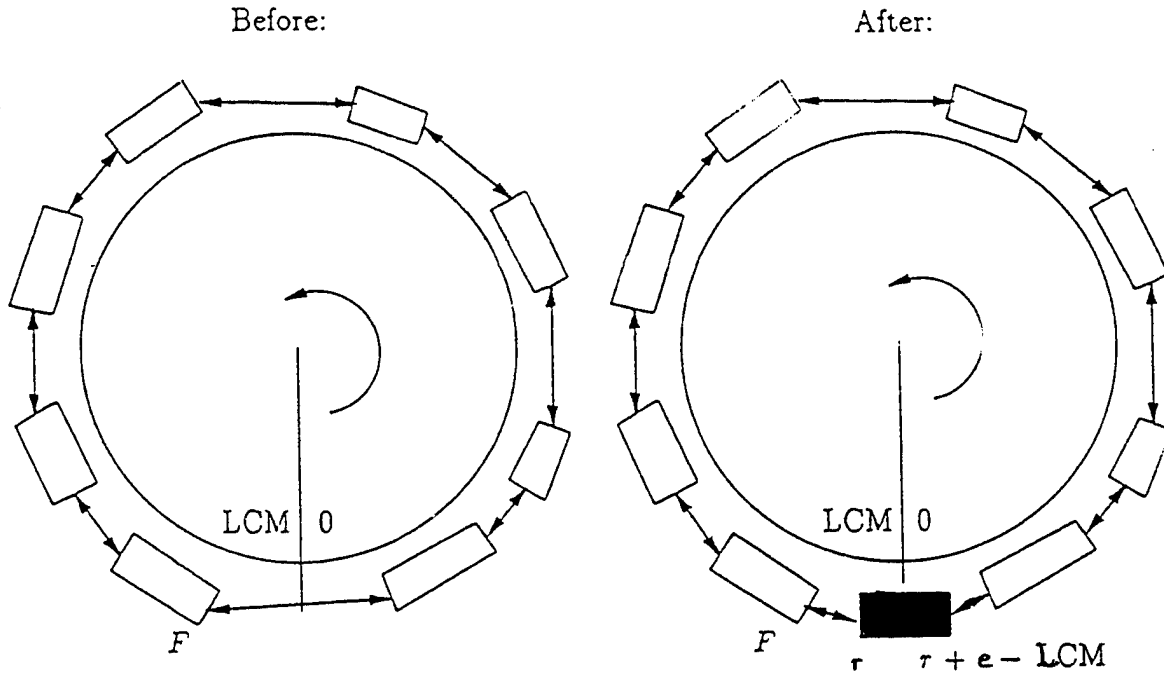
Unscheduled Instance: ▮ where $F < \tau < \text{LCM}$

Figure 3: Insertion of a new time slot

3.2.1 Recurrence

Given any solution point $(\phi, \sigma_m, \sigma_c)$, we construct the schedule by inserting time slots to the linked lists. Let σ_m: $task\_id \times instance\_id \rightarrow integer$. The insertion of a time slot for τ_i^j precedes that for τ_k^ℓ if $\sigma_m(\tau_i^j) < \sigma_m(\tau_k^\ell)$.

Recall that Equations 6 and 7 specify the bounds of the scheduling window for a task instance. Due to the communications, $est(\tau_i^j)$ in Equation 6 may not be the earliest time for τ_i^j. We define the *effective start time* as the time when (1) the hybrid constraints are satisfied and (2) τ_i^j receives all necessary data or messages from all the senders.

Given the effective start time τ and the assignment of τ_i (i.e. $p = \phi(\tau_i)$), a time slot of processor p is assigned to τ_i^j where $start\_time \geq \tau$ and $finish\_time - start\_time = e_i$. *Note* that we have to make sure the new time slot does not overlap existent time slots. Since (1) the executions of all instances within one scheduling frame recur in the next scheduling frame and (2) it is possible that the time slot for some instance is over LCM, we subtract one LCM from the $start\_time$ or $finish\_time$ if it is greater than LCM. It means the time slot for this task instance will be modulated

Equations 3 and 4. Formally, given s_i^1, s_i^2, ..., and ..., s_i^{j-1}, the problem is to derive the feasible scheduling window for τ_i^j such that a feasible schedule can be obtained if τ_i^j is scheduled within the window.

Proposition 1 [CA93]: Let the est and lst of τ_i^j be

$$est(\tau_i^j) \;=\; max\{(s_i^{j-1} + p_i - \lambda_i),\; (s_i^1 + (j-1) \times p_i - (n_i - j + 1) \times \eta_i)\}, \tag{6}$$

$$\text{and } lst(\tau_i^j) \;=\; min\{(s_i^{j-1} + p_i + \eta_i),\; (s_i^1 + (j-1) \times p_i + (n_i - j + 1) \times \lambda_i)\}. \tag{7}$$

If s_i^j is in between the $est(\tau_i^j)$ and $lst(\tau_i^j)$, then the estimated est and lst of $s_i^{n_i}$, based on s_i^j and $s_i^{n_i+1}$, specify a feasible window.

3.2 Cyclic Scheduling Technique

The basic approach of scheduling a set of synchronous periodic tasks is to consider the execution of all instances within the scheduling frame whose length is the LCM of all periods. The release times of the first periods of all tasks are zero. As long as one instance is scheduled in each period within the frame and these executions meet the timing constraints, a feasible schedule is obtained. In a feasible schedule, all instances complete the executions before the LCM.

On the other hand, in asynchronous task systems, as depicted in Figure 2 in which the LCM is $200ms$, the periods of the two tasks are out of phase. It is possible that the completion time of some instance in a feasible schedule exceeds the LCM. To find a feasible schedule for such an asynchronous system, a technique of handling the time value which exceeds the LCM is proposed.

The technique is based on the linked list structure described in the work [CA93]. Without loss of generality, we assume the minimum release time among the first periods of all tasks is zero. We keep a linked list for each processor and a separated list for the communication network. Each element in the list represents a time slot assigned to some instance or communication. The fields of a time slot of some processor p: (1) *task id i* and *instance id j* indicate the identifier of the time slot. (2) *start time st* and *finish time ft* indicate the start time and completion time of τ_i^j respectively. (3) *prev ptr* and *next ptr* are the pointers to the preceding and succeeding time slots respectively. The list is arranged in an increasing order of *start_time*. Any two time slots are nonoverlapping. Since the execution of an instance is nonpreemptable, the time difference between *start_time* and *finish_time* equals the execution time of the task.

114
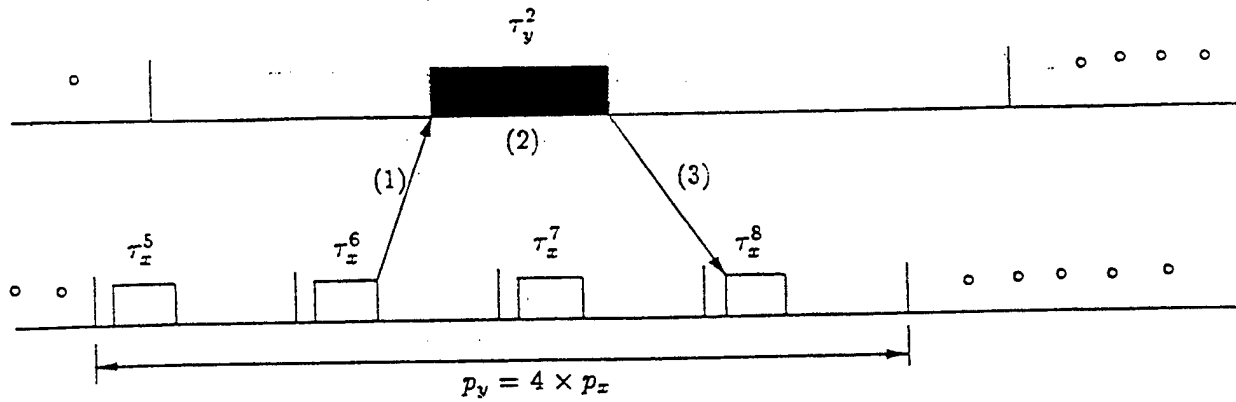
Figure 5: Asynchronous communications in mutuality

and k can be stated as

$$(j-1) \times n < i < k \leq j \times n. \tag{8}$$

A graphical illustration can be found in Figure 5. In the example, the values of i, j, k, and n are 6, 2, 8, 4 respectively. The communications $\tau_x^6 \mapsto \tau_y^2$ and $\tau_y^2 \mapsto \tau_x^8$ are scheduled before and after the scheduling of τ_y^2 respectively.

4 The Simulated Annealing Algorithm

Kirkpatrick et al. [KGV83] proposed a simulated annealing algorithm for combinatorial optimization problems. Simulated annealing is a global optimization technique. It is derived from the observation that an optimization problem can be identified with a fluid. There exists an analogy between finding an optimal solution of a combinatorial problem with many variables and the slow cooling of a molten metal until it reaches its low energy ground state. Hence, the terms about energy function, temperature, and thermal equilibrium are mostly used. During the search of an optimal solution, the algorithm always accepts the downward moves from the current solution point to the points of lower energy values, while there is still a small chance of accepting upward moves to the points of higher energy values. The probability of accepting an uphill move is a function of current temperature. The purpose of hill climbing is to escape from a local optimal configuration. If there are no upward or downward moves over a number of iterations, the thermal equilibrium is reached. The temperature then is reduced to a smaller value and the searching continues from the current solution point. The whole process terminates when either (1) the lowest energy point is found or (2) no upward or downward jumps have been taken for a number of successive thermal equilibrium.

The structure of simulated annealing (SA) algorithm is shown in Figure 7. The first step of
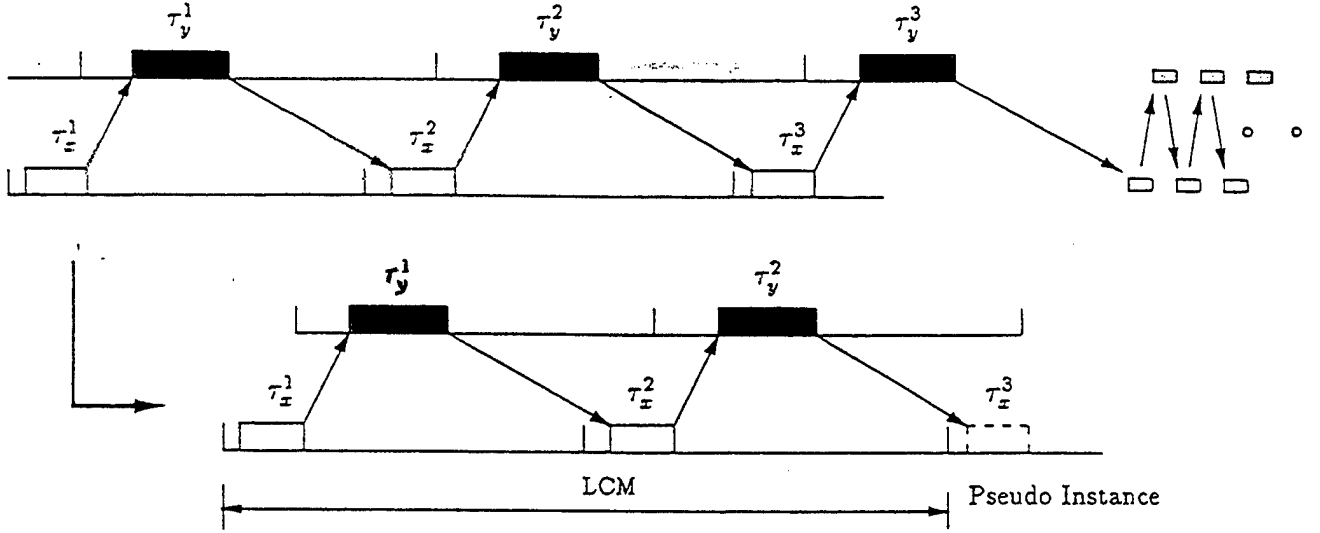
Figure 4: The introduction of a pseudo instance

and wrapped to the beginning of the schedule. As shown in Figure 3 The *start_time* of the new slot is r while the completion time is $r + e - \text{LCM}$.

3.3 Pseudo Instances

As stated in Section 2, we consider the communication pattern in which cyclic dependency exists among tasks. Given a set of tasks, Γ, a set of task instances, I, a set of communications, C, and any solution point, $(\phi, \sigma_m, \sigma_c)$, we introduce pseudo instances to solve this problem. For any task τ_x, if there exists a task τ_y, in which (1) $\sigma_m(\tau_x^i) < \sigma_m(\tau_y^i)$, $\forall\ i$, (2) $n_x = n_y$, and (3) $\tau_x \mapsto \tau_y \in C$ and $\tau_y \mapsto \tau_x \in C$, then a pseudo instance $\tau_x^{n_x+1}$ is added to I. A pseudo instance is always a receiving instance. No insertion of time slots for pseudo instances is needed. For a pseudo instance, only the effective start time is concerned. The effective start time of a pseudo instance $\tau_x^{n_x+1}$ in the constructed schedule based on $(\phi, \sigma_m, \sigma_c)$ is checked to see whether it is less than $\text{LCM} + s_x^1$ or not. If yes, then the execution of τ_x^1 for the next scheduling frame may start at $LCM + s_x^1$ which is exactly one LCM away from the execution of τ_x^1 for the current scheduling frame. A graphical illustration of the introduction of pseudo instance to solve the synchronous communications of cyclic dependency is given in Figure 4 in which $n_x = 2$.

As for the asynchronous communications of cyclic dependency, no pseudo instances are needed. For example, if both $\tau_x \mapsto \tau_y$ and $\tau_y \mapsto \tau_x$ exist and $n_x = n_y \times n$, then for each τ_y^j, where $j = 1$, $2, \ldots, n_y$, find a sending instance $\tau_x^i \in I$ and a receiving instance $\tau_x^k \in I$ such that (1) $f_x^i \leq s_y^j$, (2) $f_y^j \leq s_x^k$, and (3) $\tau_x^i \mapsto \tau_y^j$ and $\tau_y^j \mapsto \tau_x^k$ are the communications. The relationship between i, j,

116

4.1.1 Priority Assignment of Task Instances: σ_m

In the work [CA93], we presented the SLsF algorithm and the performance evaluation. The results showed that SLsF outperforms SPF and SJF. In this paper we use the SLsF as the priority assignment algorithm for the task instances in I.

Formally, if $lst(\tau_i^j) < lst(\tau_k^\ell)$, then $\sigma_m(\tau_i^j) < \sigma_m(\tau_k^\ell)$. And the insertion of a time slot for τ_i^j precedes that for τ_k^ℓ if $\sigma_m(\tau_i^j) < \sigma_m(\tau_k^\ell)$. The time-based scheduling algorithm for a task instance is used to find a time slot for a task instance once the *effective start time* is given. We define the *effective start time* of a task instance as the earliest start time when the incoming communications are taken into account. Let t be the maximum completion time among all the incoming communications of a task instance, then the *effective start time* of the task instance is set to the bigger value among t and *est* (as stated in Equation 6).

4.1.2 Scheduling the Incoming Communications: σ_c

There are two kinds of incoming communications. The first kind is called the synchronous communication in which the frequencies of the sender and receiver are identical. The other kind is called the asynchronous communication in which the sending task instance is associated with a question mark. For such an asynchronous communication, we have to decide which instance of the sending task should communicate with the receiving task instance. The approach we take is to find the nearest instance of the sending task. The reason is that, by finding the nearest instance, the time difference between start time of the receiving instance and the completion time of the sending instance is the smallest. The chance of violating the latency constraint of a communication will be the smallest then.

The nearest instance of a sending task can be found using the following method. Given an incoming communication $\tau_k^? \longrightarrow \tau_i^j$, and the *effective start time* of τ_i^j, *eft* we search through the linked list of processor $\phi(\tau_k)$ up to time *eft*. If there is some instance of τ_k, say τ_k^ℓ, whose completion time is the latest among all scheduled instances of τ_k, then the nearest instance is found. Otherwise, we continue to search through the linked list until an instance of τ_k is found. We set the *effective start time* of the communication to be the completion time of the found instance. We also erase the question mark such that $\tau_k^? \longrightarrow \tau_i^j$ is changed to $\tau_k^\ell \longrightarrow \tau_i^j$. For the synchronous communication, the *effective start time* of the communication is simply assigned as the *finish time* of the sending task instance.

The scheduling of the communication is done by inserting a time slot to the linked list for the communications network. The *start time* of the time slot can not be earlier than the *effective start*

the algorithm is to randomly choose an assignment ϕ, a total ordering of instances within one scheduling frame, σ_m, and a total ordering of communications for the instances, σ_c. A solution point in the search space of SA is a 3-tuple $(\phi, \sigma_m, \sigma_c)$. The energy of a solution point is computed by equation (5). For each solution point P which is infeasible, (i.e. E_p is nonzero), a neighbor finding strategy is invoked to generate a neighbor of P. As stated before, if the energy of the neighbor is lower than the current value, we accept the neighbor as the current solution; otherwise, a probability function (i.e. $exp(\frac{E_p - E_n}{T})$) is evaluated to determine whether to accept the neighbor or not. The parameter of the probability function is the current temperature. As the temperature is decreasing, the chance of accepting an uphill jump (i.e. a solution point with a higher energy level) is smaller. The inner and outer loops are for thermal equilibrium and termination respectively. The number of iterations for the inner loop is also a function of current temperature. The lower the temperature is, the bigger the number is. Methods about how to model the numbers of iterations and how to assign the number for each temperature have been proposed [LH91]. In this dissertation, we consider a simple incremental function. Namely, $N = N + \Delta$ where N is the number of iterations and Δ is a constant. The termination condition for the outer loop is $E_p = 0$. Whenever thermal equilibrium is reached at a temperature, the temperature is decreased. Linear or nonlinear approach of temperature decrease function can be simple or complex. Here we consider a simple multiplication function (i.e. $T = T \times \alpha$, where $\alpha < 1$).

4.1 Evaluation of Energy Value for a Solution Point $(\phi, \sigma_m, \sigma_c)$

The computation of the energy value stated in Equation 5 , is done by constructing multi-processor schedules and a network schedule, and collecting the the start and completion times of each task instance and communication from these schedules.

The construction of the schedules is characterized by the priority assignment of the task instances in the set. The priority assignment algorithm determines the scheduling order among all the task instances. Each time when a task instance is chosen to be scheduled, the incoming communications of the instance are scheduled first and then the task instance itself. After all the task instances have been scheduled, the scheduling of the outgoing communications is performed. An algorithmic description about how to compute the energy value for a solution point is given in Figure 6. Note that a communication is an incoming communication to a task instance if the frequency of the receiving task instance is equal to or less than that of the sending task instance. For example, $\tau_k^? \mapsto \tau_i^j$ and $\tau_k^j \mapsto \tau_i^j$ are incoming communications to τ_i^j. On the other hand, if the sender frequency is less than the receiver frequency, then the communication is an outgoing communication. (e.g. $\tau_k^j \mapsto \tau_i^?$ is the outgoing communication of τ_k^j).

time of the communication. Once the time slot is inserted, we check the *effective start time* of τ_i^j to make sure that it is not less than the *finish time* of the time slot. If it is, the *effective start time* of τ_i^j is updated to be the *finish time* of the time slot.

If a task instance has more than one incoming communication, the scheduling order among these communications is based on their latency constraints. The bigger the latency value is, the earlier the communication is scheduled. The incoming communication with the tightest latency constraint is scheduled last. It is because the *effective start time* of the receiving task instance is constantly updated by the scheduling of the incoming communications. It is possible that the scheduling of the later incoming communications increases the *effective start time* of the receiving task instance and make the early scheduled communication violate its latency constraint if the constraint is tight.

4.1.3 Scheduling the Outgoing Communications: σ_c

The scheduling of the outgoing communications for the whole task set is performed after all the task instances have been scheduled. The scheduling order among these communications is based on the finish times of the sending task instances. The task instance with the smallest finish time is considered first. When a task instance is taken into account, all its outgoing communications are scheduled one by one according to their latency constraints. The communication with the tightest latency constraint is scheduled first.

Given an outgoing communication $\tau_i^j \mapsto \tau_k^7$, and the finish time of τ_i^j, f_i^j, the *effective start time* of the communication is set to be f_i^j. Based on the *effective start time*, a time slot in inserted for this communication. Then the nearest instance of receiving task can be found based on the *finish time* of the time slot.

For the example shown in Figure 5, The incoming communication marked with "(1)" is scheduled before the scheduling of τ_y^2. The sixth instance of τ_z is chosen as the nearest instance. As for the outgoing communication marked with "(3)", it is scheduled after the scheduling of τ_z^5, τ_z^6, τ_z^7, and τ_z^8. In this example, τ_x^8 is the nearest instance of the outgoing communication.

4.2 Neighbor Finding Strategy: ϕ

The neighbor finding strategy is used to find the next solution point once the current solution point is evaluated as infeasible (i.e. energy value is nonnegative). The neighbor space of a solution point is the set of points which can be reached by changing the assignment of one or two tasks. There are several modes of neighbor finding strategy.

- Balance Mode: We randomly move a task from the heavily-loaded processor to the lightest-loaded processor. This move tries to balance the workload of processors. By balancing the workload, the chance to find a neighbor with a lower energy value is bigger.

- Swap Mode: We randomly choose two tasks τ_i and τ_j on processors p and q respectively. Then we change ϕ by setting $\phi(\tau_i) = q$ and $\phi(\tau_j) = p$.

- Merge Mode: We pick two tasks and move them to one processor. By merging two tasks to a processor, we increase the workload of the processor. There is an opportunity of increasing the energy level of the new point by increasing the workload of the processor. The purpose of the move is to perturb the system and allow the next move to escape from the local optimum.

- Direct Mode: When the system is in a low-energy state, only few tasks violate the jitter or latency constraints. Under such a circumstance, it will be more beneficial to change the assignment of these tasks instead of randomly moving other tasks. From the conducted experiments, we find that this mode can accelerate the searching of a feasible solution especially when the system is about to reach the equilibrium.

The selection of the appropriate mode to find a neighbor is based on the current system state. Given a randomly generated initial state (i.e. solution point), the workload discrepancy between the processors may be huge. Hence, in the early stage of the simulated annealing, the balance mode is useful to balance the workload. After the processor workload is balanced out, the swap mode and the merge mode are frequently used to find a lower energy state until the system reaches near-termination state. In the final stage of the annealing, the direct mode tries to find a feasible solution. The whole process terminates when a feasible solution is found in which the energy value is zero.

5   Experimental Results

We implemented the algorithm as the framework of the allocator on *MARUTI* [GMK+91, MSA92, SdSA94], a real-time operating system developed at the University of Maryland, and conducted extensive experiments under various task characteristics. The tests involve the allocation of real-time tasks on a homogeneous distributed system connected by a communication channel.

To test the practicality of the approach and show the significance of the algorithm, we consider a simplified and sanitized version of a real problem. This was derived from actual development work, and is therefore representative of the scheduling requirements of an actual avionics system. The Boeing 777 Aircraft Information Management System (AIMS) is to be running on a multiprocessor

	10_Proc	9_Proc	8_Proc	7_Proc	6_Proc
Exec_Time (Sec)	2369	5572	19774	36218	78647
$= Hr : Min : Sec$	0:39:29	1:32:52	5:29:34	10:03:38	21:50:47

Table 1: The execution times of the AIMS with different number of processors

system connected by a SafeBus (TM) ultra-reliable bus. The problem is to find the minimum number of processors needed to assign the tasks to these processors. The objective is to develop an off-line non-preemptable schedule for each processor and one schedule for the SafeBus (TM) ultra-reliable bus.

The AIMS consists of 155 tasks and 951 communications between these tasks. The frequencies of the tasks vary from 5HZ to 40HZ. The execution times of the tasks vary from $0ms$ to $16.650ms$. The NEI and XEI of a task t_i are $p_i - 500\mu s$ and $p_i + 500\mu s$ respectively. Since $\delta = 1000\mu s = 1ms < \frac{25ms}{e}$, the smallest-period-first scheduling algorithm can be used in this case. Tasks communicate with others asynchronously and in mutuality. The transmission times for communications are in the range from $0\mu s$ to $447.733\mu s$. The latency constraints of the communications vary from $68.993ms$ to $200ms$. The LCM of these 155 tasks is $200ms$. When the whole system is extended, the total number of task instances within one scheduling frame is 624 and the number of communications is 1580.

For such a real and tremendous problem size, pre-analysis is necessary. We calculate the resource utilization index to estimate the minimum number of processors needed to run AIMS. The index is defined as

$$\frac{\sum_{i=1}^{155}(e_i \times q_i)}{LCM}$$

where e_i is the execution of task t_i and $q_i = \frac{LCM}{p_i}$. The obtained index for AIMS is 5.14. It means there exist no feasible solutions for the AIMS if the number of processors in the multiprocessor system is less than 6.

The number of processors which the AIMS is allowed to run on is a parameter to the scheduling problem. We start the AIMS scheduling problem with 10 processors. After a feasible solution is found, we decrease the number of processors by one and solve the whole problem again. We run the algorithm on a DECstation 5000. The execution time for the AIMS scheduling problem with different numbers of processors is summarized in Table 1. The algorithm is able to find a feasible solution of the AIMS with six processors which is the minimum number of processors according to the resource utilization index. The time to find such a feasible solution is less than one day (approximately 22 hours).

5.1 Discussions

For feasible solutions of the AIMS with various numbers of processors, we calculate the processor utilization ratio (PUR) of each processor. The processor utilization ratio for a processor p is defined as

$$\frac{\sum_{\phi(t_i)=p}(e_i \times q_i)}{LCM}.$$

The results are shown in Figure 8. The ratios are sorted into a non-decreasing order given a fixed number of processors. The algorithm generates the feasible solutions for the AIMS with 6, 7, 8, 9 and 10 processors respectively. For example, for the 6-processor case, the PURs for the heaviest-loaded and lightest-loaded processors are 0.91 and 0.76 respectively. For the 10-processor cases, the PURs are 0.63 and 0.28 respectively. We find that the ratio difference between the heaviest-loaded processor and the lightest-loaded processor in the 6-processor case is smaller than those in other cases. It means the chance for a more load-balanced allocation to find a feasible solution is bigger when the number of processors is smaller.

The detailed schedules for the 6-processor case are shown in Figure 9. The results are shown on an interactive graphical interface which is developed for the design of *MARUTI*. The time scale shown in Figure 9 is $100\mu s$. So the LCM is shown as 2000 in the figure. (i.e. $2000 \times 100\mu s =$ $200ms$.) This solution consists of seven off-line non-preemptive schedules: one for each processor and one for the SafeBus (TM). Each of these schedules will be one LCM long where an infinite schedule can be produced by repeating these schedules indefinitely. Note that the pseudo instances are introduced to make sure the wrapping around at the end of the LCM-long schedules should satisfy the latency and next-execution-interval requirements across the point of wrap-around. The pseudo instances are not shown in Figure 9.

The inclusion of resource and memory constraints into the problem can be done by modifying neighbor-finding strategy. Once a neighbor of the current point is generated, it is checked to ascertain that the constraints on memory etc. are met. If not, the neighbor is discarded and another neighbor is evaluated.

References

[CA93] Sheng-Tzong Cheng and Ashok K. Agrawala. Scheduling of periodic tasks with relative timing constraints. Technical Report CS-TR-3392, UMIACS-TR-94-135, Department of Computer Science, University of Maryland, College Park, December. 1993. Submitted to *the 10th Annual IEEE Conference on Computer Assurance, COMPASS '95*.

[CDHC94] T. Carpenter, K. Driscoll, K. Hoyme, and J. Carciofini. Arinc 659 scheduling: Problem definition. In *Proceedings of IEEE Real-Time Systems Symposium*, San Juan, PR, Dec. 1994.

[GMK+91] Ó. Gudmundsson, D. Mossé, K.T. Ko, A.K. Agrawala, and S.K. Tripathi. Maruti: A platform for hard real-time applications. In K. Gordon, A.K. Agrawala, and P. Hwang (eds.), editors, *Mission Critical Operating Systems*. IOS Press, 1991.

[HS92] Chao-Ju Hou and Kang G. Shin. Allocation of periodic task modules with precedence and deadline constraints in distributed real-time systems. In *Proceedings of the 1992 IEEE 13th Real-Time Systems Symposium*, pages 146–155, Phoenix, AZ, 1992.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[LH91] Feng-Tse Lin and Ching-Chi Hsu. Task assignment problems in distributed computing systems by simulated annealing. *Journal of the Chinese Institute of Engineers*, 14(5):537–550, Sept. 1991.

[MSA92] Daniel Mossé, M.C. Saksena, and Ashok K. Agrawala. Maruti: An approach to real-time system design. Technical Report CS-TR-2845, UMIACS-TR-92-21, Department of Computer Science, University of Maryland, College Park, 1992.

[Ram90] Krithi Ramamritham. Allocation and scheduling of complex periodic tasks. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 108–115, Paris, France, 1990.

[SdSA94] M. Saksena, J. da Silva, and A. K. Agrawala. Design and implementation of maruti-ii. Technical Report CS-TR-2845, Department of Computer Science, University of Maryland, College Park, 1994.

[TBW92] K. W. Tindell, A. Burns, and A. J. Wellings. Allocating hard real-time tasks: an NP-hard problem made easy. *Real-Time Systems*, 4(2):145–165, June 1992.

```
Given a solution point $P = (\phi, \sigma_m, \sigma_c)$
While there is some unscheduled task instance do
      Find the next unscheduled instance. /* By the SLsF algorithm */
      Let the instance be $\tau_i^j$.
      Sort all the incoming communications of $\tau_i^j$ based on
            the latency values into a descending order.
      Schedule each incoming communication starting from
            the biggest-latency one to the tightest-latency one.
      Schedule the instance $\tau_i^j$.
End While.
Mark each instance as un-examined.
While there is some un-examined task instance do
      Find the next un-examined task instance. /* By the finish times */
      Sort all the outgoing communications of the task instance based
            on the latency values into an increasing order.
      Schedule each outgoing communication starting from
            the tightest-latency one to the biggest-latency one.
      Mark the task instance examined.
End While.
Collect the start time and finish time informations for each task instance and communication.
Compute the energy value using Equation 5.
```

Figure 6: The pseudo code for computing the energy value

124

```
Choose an initial temperature $T$
Choose randomly a starting point $P = (\phi, \sigma_m, \sigma_c)$
$E_p :=$ Energy of solution point $P$
if $E_p = 0$ then
        output $E_p$ and exit /* $E_p = 0$ means a feasible solution */
end if
repeat
      repeat
              Choose $N$, a neighbor of $P$
              $E_n :=$ Energy of solution point $N$
              if $E_n = 0$ then
                      output $E_n$ and exit /* $E_n = 0$ means a feasible solution */
              end if
              if $E_n < E_p$ then
                      $P := N$
                      $E_p := E_n$
              else
                      $x := \frac{E_p - E_n}{T}$
                      if $e^x \geq$ random(0,1) then
                              $P := N$
                              $E_p := E_n$
                      end if
              end if
      until thermal equilibrium at $T$
      $T := \alpha \times T$ (where $\alpha < 1$)
until stopping criterion
```

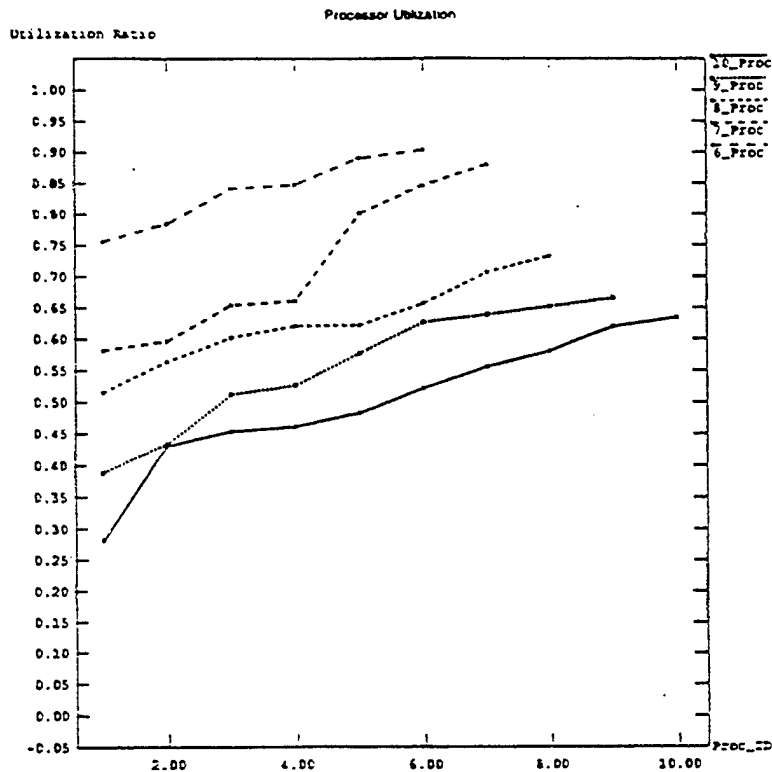Figure 7: The structure of simulated annealing algorithm.
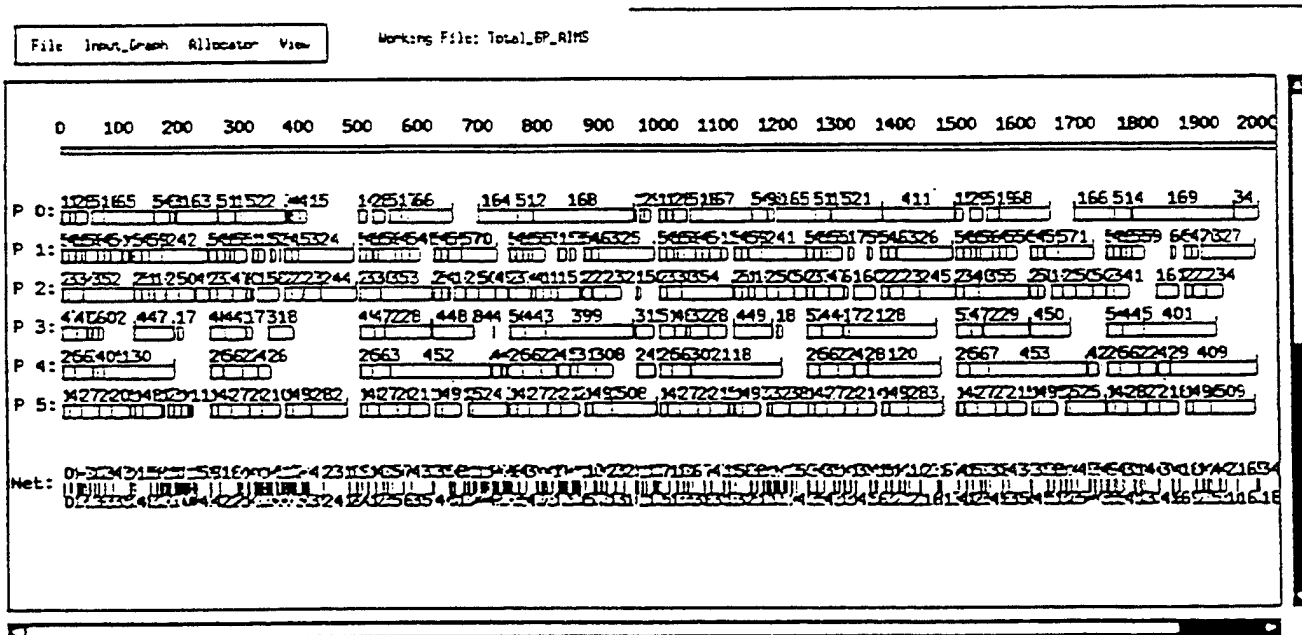
Figure 8: Processor Utilization Ratios for different cases

Figure 9: The Allocation Results and Schedules for AIMS with 6 processors

126

REPORT DOCUMENTATION PAGE

Form approved
OMB No 074-0188

1. AGENCY USE ONLY (leave blank)	2. REPORT DATE January 1995	3. REPORT TYPE END DATES COVERED Technical Report

4. TITLE AND SUBTITLE
Allocation and Scheduling of Real-Time Periodic Tasks with Relative Timing Constraints

5. FUNDING NUMNBERS
N00014091-C-0195
DSAG-60-C-0055

6. AUTHOR(S)
Sheng-Tzong Cheng and Ashok K. Agrawala

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Maryland
Department of Computer Science
A.V. Williams Building
College Park, MD 20742

8. PERFORMING ORGANIZATION REPORT NUMBER
CS-TR-3402
UMIACS-TR-95-6

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Honeywell Inc.
3600 Technology Drive
Minneapolis, MN 55148

Phillips Laboratory
Directorate of Contracting
3651 Lowry Avenue, SE
Kirtland AFB, NM 87117-5777

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12.a. DISTRIBUTION/ AVAILABILITY STATEMENT

12.b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Allocation problem has always been one of the fundamental issues of building the applications in distributed computing systems (DCS). For real-time applications on DCS, the allocation problem should directly address the issues of task and communication scheduling. In this context, the allocation of tasks has to fully utilize the available processors and the scheduling of tasks has to meet the specified timing constraints. Clearly, the execution of tasks under the allocation and schedule has to satisfy the precedence, resources, and other synchronization constraints among them.

Recently, the timing requirements of the real-time systems emerge that the relative timng constraints are imposed on the consecutive executions of each task and the inter-task temporal relationships are specified across task periods. In this paper we consider the allocation and scheduling problem of the periodic tasks with such timing requirements. Given a set of periodic tasks, we consider the least common multiple (LCM) of the task periods. Each task is extended to several instances within the LCM. The scheduling window for each task instance is derived to satisfy the timing constraints. We develop a simulated annealing algorithm as the overall control algorithm. An example problem of the sanitized version of the Boeing 777 Aircraft Information Management Systems is solved by the algorithm. Experimental results show that the algorithm solves the problem in a reasonable time complexity.

14. SUBJECT TERMS
Process Management; Special Purpose and Application-Based Systems

15. NUMBER OF PAGES
22

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlisted

Scheduling of Periodic Tasks with Relative Timing Constraints *

Sheng-Tzong Cheng and Ashok K. Agrawala
Institute for Advanced Computer Studies
Systems Design and Analysis Group
Department of Computer Science
University of Maryland
College Park, MD 20742
{stcheng,agrawala}@cs.umd.edu

Abstract

The problem of non-preemptive scheduling of a set of periodic tasks on a single processor has been traditionally considering the ready time and deadline on each task. As a consequence, a feasible schedule finds that in each period one instance of each task starts the execution after the ready time and completes the execution before the deadline .

Recently, the timing requirements of the real-time systems emerge that the relative timing constraints are imposed on the consecutive executions of each task. In this paper, we consider the scheduling problem of the periodic tasks with the relative timing constraints imposed on two consecutive executions of a task. We analyze the timing constraints and derive the scheduling window for each task instance. Based on the scheduling window, we present the time-based approach of scheduling a task instance. The task instances are scheduled one by one based on their priorities assigned by the proposed algorithms in this paper. We conduct the experiments to compare the schedulability of the algorithms.

1 Introduction

The task scheduling problem is one of the basic issues of building real-time applications in which the tasks of applications are associated with timing constraints. For the hard real-time applications, such as avionics systems and nuclear power systems, the approach to guarantee the critical timing constraints is to schedule periodic tasks *a priori*. A non-preemptive schedule for a set of periodic tasks is generated by assigning a start time to each execution of a task to meet their timing constraints. Failure to meet the specified timing constraints can result in disastrous consequence.

Various kinds of periodic task models have been proposed to represent the real-time system characteristics. One of them is to model an application as a set of tasks, in which each task is executed once every period under the ready time and deadline constraints. These constraints impose constant intervals in which a task can be executed. In literature, many techniques [2, 3, 4, 5, 6, 7, 8] have been proposed to solve the scheduling problem in this context. The deficiency of this modeling is the inability of specifying the relative constraints across task periods. For example, one can not specify the timing relationship between two consecutive executions of the same task.

Simply assuring that one instance of each task starts the execution after the ready time and completes the execution before the specified deadline is not enough. Some real-time applications have more complicated timing constraints for the tasks. For example, the relative timing constraints may be imposed upon the consecutive executions of a task in which the scheduling of two consecutive executions of a periodic task must be separated by a minimum execution interval. The Boeing 777 Aircraft Information Management System is such an example [1]. One possible solution to the scheduling problem of such applications is to consider the instances of tasks rather than the tasks. A task instance is defined as one execution of a task within a period. With the notion of task instances, one is able to specify the various timing constraints and dependencies among instances of tasks.

In this paper, we consider the relative timing constraints imposed on two consecutive instances of a task. The task model and the analysis of the timing constraints are introduced in Sections 2 and 3 respectively. Based on the analysis, we are able to derive the scheduling window for each task instance. Given the scheduling window of a task instance, we present the time-based approach of scheduling a task instance in Section 4. We propose three priority assignment algorithms for the task instances in Section 5. The task instances are scheduled one by one based on their priorities. In Section 6, we evaluate the three algorithms and show the experimental results.

2 Problem Statement

Consider a set of periodic tasks $\Gamma = \{ \tau_i \mid i = 1, \ldots n \}$, where τ_i is a 4-tuple $< p_i, e_i, \lambda_i, \eta_i >$ denoting the period, computation time, low jitter and high jitter respectively. One instance of a task is executed each period. The execution of a task instance is non-preemptable. The start times of two consecutive instances of task τ_i are at least $p_i - \lambda_i$ and at most $p_i + \eta_i$ apart.

In order to schedule periodic tasks, we consider the least common multiple (LCM) of all periods of tasks. Let n_i be the number of instances for task τ_i within a schedule of length LCM. Hence, $n_i = \frac{LCM}{p_i}$. A schedule for a set of tasks is the mapping of each task τ_i to n_i task instances and the assigning of a start time s_i^j to the j-th instance of task τ_i, τ_i^j, $\forall\ i = 1, \ldots n$ and $j = 1, \ldots, n_i$. A *feasible* schedule is a schedule in which the following conditions are satisfied for each task τ_i:

$$f_i^j = s_i^j + e_i \tag{1}$$

$$s_i^{n_i+1} = s_i^1 + LCM \tag{2}$$

$$s_i^j \geq s_i^{j-1} + p_i - \lambda_i \tag{3}$$

$$s_i^j \leq s_i^{j-1} + p_i + \eta_i \tag{4}$$

$$\forall j = 2, \ldots, n_i + 1.$$

The non-preemption scheduling discipline leads to Equation 1 where f_i^j is the finish time of τ_i^j. Another condition for non-preemption scheduling is that given any i, j, k and ℓ, if $s_i^j < s_k^\ell$ then $f_i^j \leq s_k^\ell$. It means the schedule for any two instances is non-overlapping. The constructed schedule of length LCM is invoked repeatedly by wrapping-around the end point of the first schedule to the start point of the next one. Hence, as shown in Equation 2, the start time of the first instance in the next schedule is exactly one LCM away from that of the first schedule. Finally, Equations 3 and 4 specify the relative timing constraints between two consecutive instances of a task.

3 Analysis of Relative Timing Constraints

Define the scheduling window for a task instance as the time interval during which the task can start. Traditionally, the lower and upper bounds of the scheduling window for a task instance are called earliest start time (*est*) and latest start time (*lst*) respectively. These values are given and independent of the start times of the preceding instances.

Instance ID	$est = s_i^{j-1} + p_i - \lambda_i$	$lst = s_i^{j-1} + p_i + \eta_i$	actual start time (s_i^j)
τ_i^1	0	40	4
τ_i^2	39	49	40
τ_i^3	75	85	77
τ_i^4	112	122	113
τ_i^5	148	158	*

Table 1: An example to show the wrong setting of scheduling windows

We consider the scheduling of periodic tasks with relative timing constraints described in Equations 3 and 4. The scheduling window for a task instance is derived from the start times of its preceding instances. A *feasible* scheduling window for a task instance τ_i^j is a scheduling window in which any start time in the window makes the timing relation between s_i^{j-1} and s_i^j satisfy Equations 3 and 4. Formally, given s_i^1, s_i^2, ..., and ..., s_i^{j-1}, the problem is to derive the feasible scheduling window for τ_i^j such that a feasible schedule can be obtained if τ_i^j is scheduled within the window.

For the sake of simplicity, we assume that $r_i = 0$ and $d_i = p_i$, $\forall i$, in this section. Then, simply assigning est and lst of τ_i^j as $s_i^{j-1} + p_i - \lambda_i$ and $s_i^{j-1} + p_i + \eta_i$ respectively where $i = 1, 2, ..., n$ and $j = 1, 2, ..., n_i$, is not tight enough to guarantee a feasible solution. For example, consider the case shown in Table 1 in which a periodic task τ_i is to be scheduled. Let LCM, p_i, λ_i, and η_i be 200, 40, 5, and 5 respectively. Hence, there are 5 instances within one LCM (i.e. $n_i = 5$). The first column in Table 1 indicates the instance IDs. The second and third columns give the est and lst of the scheduling windows for the task instances specified in the first column. The last column shows the actual start times scheduled for the particular task instances. The actual start time is a value in between est and lst of each task instance. For instance, the est and lst of τ_i^2 are 39 and 49 respectively. It means $39 \leq s_i^2 \leq 49$. The scheduled value for s_i^2, in the example, is 40. Since $s_i^6 = s_i^1 + LCM = 204$, we find that any value in the interval [148,158] can not satisfy the relative timing constraints between τ_i^5 and τ_i^6. As a consequence, the constructed schedule is infeasible.

We draw a picture to depict the relations among the start times of task instances in Figure 1. When τ_i^j is taken into account, the scheduling window for s_i^j is obtained by considering its relation with s_i^{j-1} as well as that with $s_i^{n_i}$ and $s_i^{n_i+1}$. We make sure that once s_i^j is determined, the estimated est and lst of $s_i^{n_i}$, based on s_i^j and $s_i^{n_i+1}$, specify a feasible scheduling window for $s_i^{n_i}$. Namely, the interval which is specified by the estimated est and lst of $s_i^{n_i}$, based on s_i^j, overlaps the interval
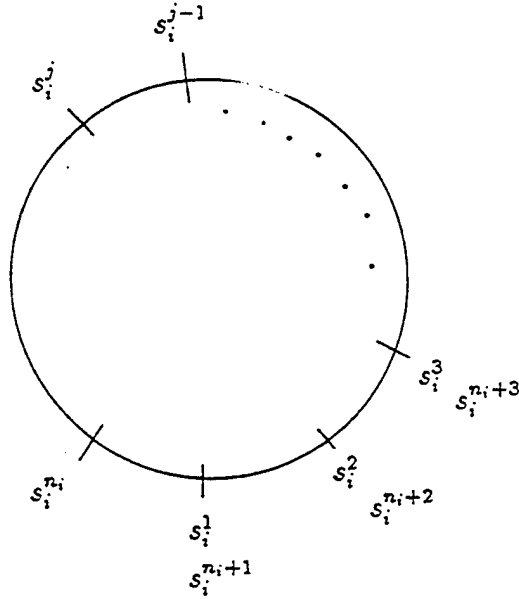
Figure 1: The relations between the task instances

$[s_i^{n_i+1} - (p_i + \eta_i), s_i^{n_i+1} - (p_i - \lambda_i)]$.

Proposition 1: Let the est and lst of τ_i^j be

$$est(\tau_i^j) \;=\; max\{(s_i^{j-1} + p_i - \lambda_i), \; (s_i^1 + (j-1) \times p_i - (n_i - j + 1) \times \eta_i)\}, \qquad (5)$$

$$\text{and } lst(\tau_i^j) \;=\; min\{(s_i^{j-1} + p_i + \eta_i), \; (s_i^1 + (j-1) \times p_i + (n_i - j + 1) \times \lambda_i)\}. \qquad (6)$$

If s_i^j is in between the $est(\tau_i^j)$ and $lst(\tau_i^j)$, then the estimated est and lst of $s_i^{n_i}$, based on s_i^j and $s_i^{n_i+1}$, specify a feasible window.

Proof: Let ℓ and μ be the estimated est and lst of $s_i^{n_i}$, based on s_i^j, respectively.

Hence,

$$\ell = s_i^j + (n_i - j) \times (p_i - \lambda_i) \qquad (7)$$

$$\mu = s_i^j + (n_i - j) \times (p_i + \eta_i) \qquad (8)$$

To guarantee the existence of feasible start time of $\tau_i^{n_i}$, the interval $[\ell, \mu]$ has to overlap the interval $[s_i^{n_i+1} - (p_i + \eta_i), s_i^{n_i+1} - (p_i - \lambda_i)]$. Hence the following conditions have to be satisfied:

$$s_i^{n_i+1} - \ell \geq p_i - \lambda_i \qquad (9)$$

133

$$s_i^{n_i+1} - \mu \leq p_i + \eta_i \qquad (10)$$

By replacing ℓ in Equation 9 with $s_i^j + (n_i - j) \times (p_i - \lambda_i)$, we obtain

$$
\begin{aligned}
s_i^j &\leq s_i^{n_i+1} - (n_i - j + 1) \times (p_i - \lambda_i) \\
&= s_i^1 + LCM - (n_i - j + 1) \times (p_i - \lambda_i) \\
&= s_i^1 + n_j \times p_i - (n_i - j + 1) \times (p_i - \lambda_i) \\
&= s_i^1 + (j - 1) \times p_i + (n_i - j + 1) \times \lambda_i \qquad (11)
\end{aligned}
$$

Likewise, by replacing μ in Equation 10 with $s_i^j + (n_i - j) \times (p_i + \eta_i)$, we have

$$
\begin{aligned}
s_i^j &\geq s_i^{n_i+1} - (n_i - j + 1) \times (p_i + \eta_i) \\
&= s_i^1 + LCM - (n_i - j + 1) \times (p_i + \eta_i) \\
&= s_i^1 + (j - 1) \times p_i - (n_i - j + 1) \times \eta_i \qquad (12)
\end{aligned}
$$

So, According to Equations 12 and 3, we choose the bigger value between $(s_i^{j-1} + p_i - \lambda_i)$ and $(s_i^1 + (j - 1) \times p_i - (n_i - j + 1) \times \eta_i)$ as the *est* of τ_i^j. Similarly, according to Equations 11 and 4, we assign the smaller value of $(s_i^{j-1} + p_i + \eta_i)$ and $(s_i^1 + (j - 1) \times p_i + (n_i - j + 1) \times \lambda_i)$ as the *lst*.

$\square$

Example 3.1: To show how Proposition 3 gives a tighter bound to find feasible scheduling windows, we consider the case shown in Table 1 again. We apply Equations 5 and 6 to compute the *est* and *lst* of each instance. The results are shown in Table 2. Note that the scheduling windows for τ_i^4 and τ_i^5 are tighter than those in Table 1. As a consequence, any start time in the interval [159,160] for τ_i^5 satisfys the relative timing constraints between τ_i^5 and τ_i^6.

3.1 Property of Scheduling Windows

Define $P_i(x, y, z)$ as the predicate in which the estimated *est* and *lst* of τ_i^y, based on s_i^x and s_i^z, specify a feasible scheduling window for τ_i^y. In Proposition 3 , we prove that for any s_i^j in between $est(\tau_i^j)$ and $lst(\tau_i^j)$ as specified in Equations 5 and 6, $P_i(j, n_i, n_i + 1)$ is true.

134

Instance ID	*est* from Equation 5	*lst* from Equation 6	actual start time (s_i^j)
τ_i^1	0	40	4
τ_i^2	39	49	40
τ_i^3	75	85	77
τ_i^4	114	122	115
τ_i^5	159	160	$159 \sim 160$

Table 2: The correct setting of scheduling windows based on Proposition 3.1.

Lemma 1 *Given s_i^1, s_i^2, ..., and s_i^j, if, $\forall k = 2, ..., j$, $est(\tau_i^k) \leq s_i^k \leq lst(\tau_i^k)$ as specified in Equations 5 and 6, then $P_i(j, y, n_i + 1)$ is true, $\forall y = j + 1, j + 2, ..., n_i$.*

Proof: We prove that the estimated *est* and *lst* of τ_i^y, based on s_i^j and $s_i^{n_i+1}$, specify a feasible scheduling window, by showing that (1) the estimated scheduling window of s_i^y, based on s_i^j, is specified by the interval

$$[s_i^j + (y - j) \times (p_i - \lambda_i), s_i^j + (y - j) \times (p_i + \eta_i)], \tag{13}$$

(2) the estimated scheduling window of s_i^y, based on $s_i^{n_i+1}$, is specified by the interval

$$[s_i^{n_i+1} - (n_i - y + 1) \times (p_i + \eta_i), s_i^{n_i+1} - (n_i - y + 1) \times (p_i - \lambda_i)], \tag{14}$$

and (3) the intervals in Equations 13 and 14 overlap.

In Figure 2, we see that the necessary and sufficient conditions for the overlapping of the intervals specified in Equations 13 and 14 are

$$s_i^j + (y - j) \times (p_i - \lambda_i) \leq s_i^{n_i+1} - (n_i - y + 1) \times (p_i - \lambda_i) \tag{15}$$

$$\text{and} \quad s_i^{n_i+1} - (n_i - y + 1) \times (p_i + \eta_i) \leq s_i^j + (y - j) \times (p_i + \eta_i). \tag{16}$$

By solving the Equations 15 and 16, we obtain

$$s_i^j \leq s_i^1 + (j - 1) \times p_i + (n_i - j + 1) \times \lambda_i$$

$$\text{and} \quad s_i^j \geq s_i^1 + (j - 1) \times p_i - (n_i - j + 1) \times \eta_i.$$

The above two equations describe the same conditions as Equations 11 and 12 do. Hence, $P_i(j, y, n_i + 1)$ is true, $\forall y = j + 1, j + 2, ..., n_i$.

135
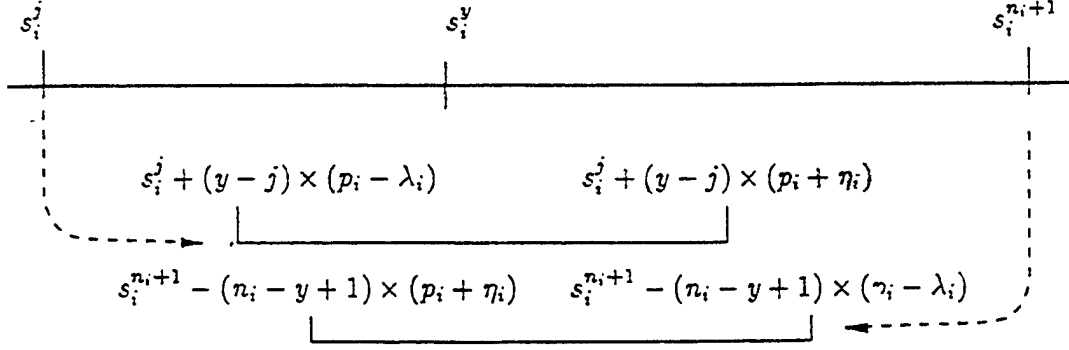
Figure 2: The overlapping of two intervals

□

Lemma 2 *Given s_i^1, s_i^2, ..., s_i^j, and an integer n_0, where $1 \leq n_0 \leq j$, if, $\forall\, k = 2, ..., j$, $est(\tau_i^k)$ $\leq s_i^k \leq lst(\tau_i^k)$ are specified as in Equations 5 and 6, then $P_i(j, y, n_i + n_0)$ is true, $\forall\, y = j + 1$, $j + 2, ..., n_i$.*

Proof: We use the same method in Lemma 1 to prove it. We show that (1) the estimated scheduling window of s_i^y, based on s_i^j, is specified by the interval

$$[s_i^j + (y - j) \times (p_i - \lambda_i), s_i^j + (y - j) \times (p_i + \eta_i)], \tag{17}$$

(2) the estimated scheduling window of s_i^y, based on $s_i^{n_i + n_0}$, is specified by the interval

$$[s_i^{n_i + n_0} - (n_i + n_0 - y) \times (p_i + \eta_i), s_i^{n_i + n_0} - (n_i + n_0 - y) \times (p_i - \lambda_i)], \tag{18}$$

and (3) these two intervals overlap.

The following conditions have to be satisfied to make sure the overlapping of the two intervals.

$$s_i^j \leq s_i^{n_0} + (j - 1) \times p_i + (n_i - j + 1) \times \lambda_i - (p_i - \lambda) \times n_0 - 1 \tag{19}$$

$$\text{and } s_i^j \geq s_i^{n_0} + (j - 1) \times p_i - (n_i - j + 1) \times \eta_i - (p_i + \eta_i) \times n_0 - 1. \tag{20}$$

Since $s_i^1 \leq s_i^{n_0} - (p_i - \lambda) \times (n_0 - 1)$ and $s_i^1 \geq s_i^{n_0} - (p_i + \eta_i) \times (n_0 - 1)$, we rewrite Equations 19 and 20

$$s_i^j \leq \underline{s_i^{n_0}} + (j - 1) \times p_i + (n_i - j + 1) \times \lambda_i - (p_i - \lambda) \times n_0 - 1$$

136

$$\leq \quad \underline{s_i^1} + (j-1) \times p_i + (n_i - j + 1) \times \lambda_i$$

$$\text{and} \quad s_i^j \quad \geq \quad \underline{s_i^{n_0}} + (j-1) \times p_i - (n_i - j + 1) \times \eta_i \underline{-(p_i + \eta_i) \times n_0 - 1}.$$

$$\geq \quad \underline{s_i^1} + (j-1) \times p_i - (n_i - j + 1) \times \eta_i$$

Hence $P_i(j, y, n_i + n_0)$ holds for any $1 \leq n_0 \leq j$.

$\square$

Theorem 1 *Given s_i^1, s_i^2, ..., and s_i^j, if, $\forall\, k = 2, ..., j$, $\text{est}(\tau_i^k) \leq s_i^k \leq \text{lst}(\tau_i^k)$ as specified in Equations 5 and 6, then $P_i(j, y, z)$ is true, $\forall\, y = j+1, j+2, ..., n_i$, and $z = n_i + 1, n_i + 2, ..., n_i + j$.*

By combining the proofs in Lemmas 1 and 2, it is easy to see that Theorem 1 holds. Based on Theorem 1 , we can assign the scheduling window for τ_i^j by using Equations 5 and 6 once s_i^1, s_i^2, ..., s_i^{j-1}.

Before we present the scheduling technique for a task instance, let us consider the following objective. The objective can be formulated as follows. Given a set of tasks with the characteristics described in Section 2, we schedule the task instances for each task within one LCM to minimize

$$\pi = \sum_{i,j} \alpha(s_i^j - s_i^{j-1} - p_i) \tag{21}$$

Subject to   the constraints specified in Equations 1 through 4,

where $\alpha(x) = x$, if $x \geq 0$; $= -x$, otherwise.

Basically, we try to schedule every instance of a task one period apart from its preceding instance. An optimal schedule is a feasible schedule with the minimum total deviation value from one period apart for instances.

4   The Time-Based Scheduling of a Task Instance

We consider the time-based solution to the scheduling problem by using a linked list. Each element in the list represents a time slot assigned to a task instance. A time slot w has the following fields: (1) *task id i* and *instance id j* indicate the identifier of the time slot. (2) *start time st* and *finish time ft* indicate the start time and completion time of τ_i^j respectively. (3) *prev ptr* and *next ptr* are the

137

new arriving instance: ▮▮▮ $[est(\tau_i^j),\ lst(\tau_i^j)]$
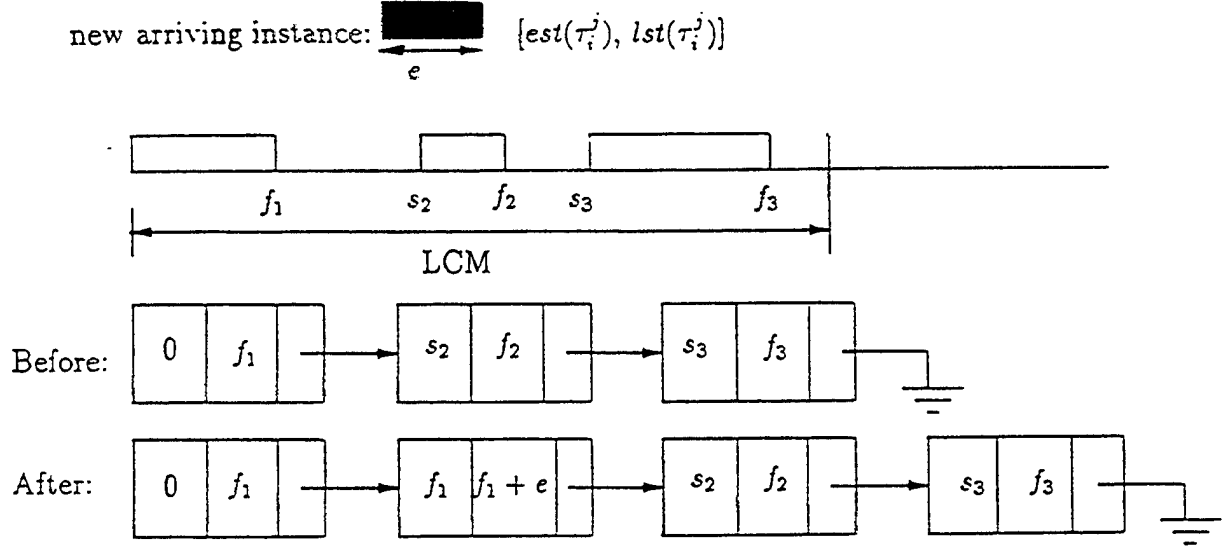
Figure 3: Insertion of a new time slot

pointers to the preceding and succeeding time slots respectively. We arrange the time slots in the list in increasing order by using the *start time* as the key. Any two time slots are non-overlapping. Since the execution of an instance is non-preemptable, the time difference between *start time* and *finish time* equals the execution time of the task.

4.1 Creating a Time Slot for the Task Instance

Consider a set of n tasks. Given a linked list and a task instance τ_i^j, we schedule the instance by inserting a time slot to the list. According to equations 5 and 6, we compute the $est(\tau_i^j)$ and $lst(\tau_i^j)$ first. Let S be the set of unoccupied time intervals that overlap the interval $[est(\tau_i^j), lst(\tau_i^j)]$ in the linked list. The unoccupied time intervals in S are collected by going through the list. Each time when a pair of time slots $(w, w+1)$ is examined, we compute $\ell = \max\{est(\tau_i^j), ft(w)\}$ and $\mu = \min\{lst(\tau_i^j), st(w+1)\}$, where $ft(w)$ is the finish time of the time slot w, and $st(w+1)$ is the start time of the slot next to w. If $\ell \le \mu$, then we add the interval $[\ell, \mu]$ to S.

The free intervals in S are the potential time slots which τ_i^j can be assigned to. Since we try to schedule τ_i^j as close to one period away from the preceding instance τ_i^{j-1} as possible, we sort S, based on the function of the lower bound of each interval, $\alpha(s_i^{j-1} + p_i - \ell)$, in ascending order. Without loss of generality, we assume that S after the sorting is denoted by $\{int_1, int_2, \ldots, int_{|S|}\}$

138

The idea is that if τ_i^j is scheduled to int_k, then the value in equation 21 will be smaller than that of the case in which τ_i^j is scheduled to int_{k+1}.

The scheduling of τ_i^j can be described as follows. Starting from int_1, we check whether the length of the interval is greater or equal to the execution time of τ_i^j or not. If yes, then we schedule the instance to the interval. One new time slot is created in which the start time is the lower bound of the interval and the finish time equals the start time plus the execution time. The created time slot is added to the linked list and the scheduling is done. If the length is smaller than the execution time, then we check the length of the next interval until all intervals are examined. An example is shown in Figure 3 in which the slot with dark area represents τ_i^j. In this example we assume that $est(\tau_i^j) \le f_1$ and $s_2 - f_1 > e$. It means the free slot between the first and second occupied slots can be assigned to τ_i^j.

4.2 Sliding of the Time Slots

In case none of the intervals in S can accommodate a task instance, the sliding technique is used to create a big enough interval by sliding the existence time slots in the list.

To make the sliding technique work, we maintain two values for each time slot: *left laxity* and *right laxity*. The value of left laxity indicates the amount of time units by which a time slot can be left-shifted to an earlier start time. Similarly, the right laxity indicates the amount of time units by which a time slot can be right-shifted to a later start time.

Given the time slots w_{k-1}, w_k, and w_{k+1}, where a and b are the task and instance identifiers of w_k respectively, the laxity values of the time slot w_k can be computed by:

$$left\_laxity(w_k) = min\{s_a^b - est', \; s_a^b - ft(w_{k-1}) + left\_laxity(w_{k-1})\} \tag{22}$$

$$right\_laxity(w_k) = min\{lst' - s_a^b, \; st(w_{k+1}) - f_a^b + right\_laxity(w_{k+1})\} \tag{23}$$

$$\text{where} \qquad est' = max\{est(\tau_a^b), \; s_a^{b+1} - (p_a + \eta_a)\}$$

$$\text{and} \qquad lst' = min\{lst(\tau_a^b), \; s_a^{b+1} - (p_a - \lambda_a)\}.$$

Note that the interval $[est', lst']$ defines the sliding range during which τ_a^b can start without shifting τ_a^{b-1} or τ_a^{b+1}. A schematic illustration of equations 22 and 23 is given in Figure 4.

From equations 22 and 23, we see that the computing of $left\_laxity(w_k)$ depends on that of w_{k-1} and the computing of $right\_laxity(w_k)$ depends on that of w_{k+1}. It implies a two-pass computation

139
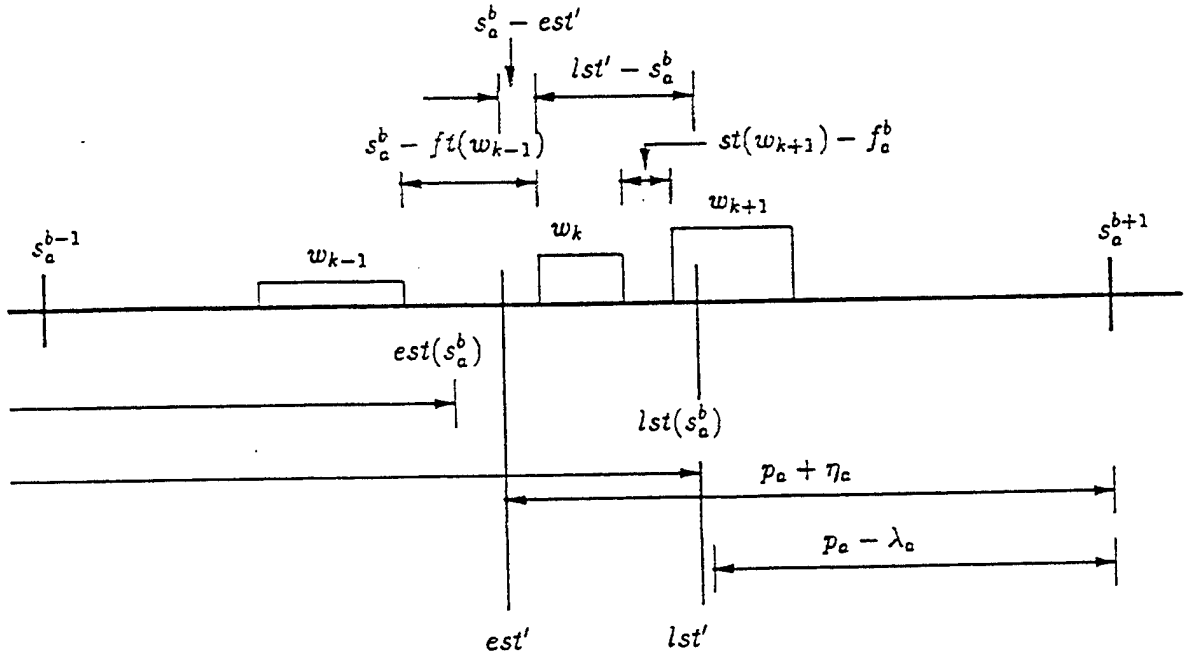
Figure 4: An illustration of $left\_laxity(w_k)$ and $right\_laxity(w_k)$

is needed to compute the laxity values for all time slots. The complexity is $O(2N)$ where N is the number of time slots in the linked list.

The basic idea of the sliding technique is described as follows. Given a task instance τ_i^j and a set of unoccupied intervals, $S = \{int_1, int_2, \ldots, int_{|S|}\}$, we check one interval at a time to see if the interval can be enlarged by shifting the existent time slots. Two possible ways of enlargement are (1) by either shifting the time slots, that precede the interval, to the left or (2) shifting the slots, that follow the interval, to the right. The shifting depends on which direction minimizes the objective function in Equation 21.

4.3 The Algorithm

An algorithmic description about how to schedule a task instance, as described in Sections 4.1 and 4.2, is given in Table 3.

The procedures Left_Shift(w_k,time_units) and Right_Shift(w_k,time_units) in Table 3 may involve the shifting of more than one time slot recursively. For example, consider the case in Figure 4, if Right_Shift(w_k,$lst' - s_a^b$) is invoked (i.e. w_k is to be shifted right by $lst' - s_a^b$ time units), then w_{k+1} has to be shifted too. It is because the gap between w_k and w_{k+1} is $st(w_{k+1}) - f_a^b$ which is

smaller than $lst' - s_a^b$. In this case, Right_Shift($w_{k+1}, lst' - s_a^b - st(w_{k+1}) + f_a^b$) is invoked.

We do not enlarge an interval at both ends. Enlarging an interval at both ends needs to shift certain amount of preceding time slots to the left and shift some succeeding slots to the right. It is possible that some task instance τ_x^y is shifted left, while τ_x^{y+1} is shifted right. As a consequence, the timing constraints between s_x^y and s_x^{y+1} could be violated. For example, Let s_x^y and s_x^{y+1} before the shifting be 10 and 20 respectively. The execution time for τ_x is 5 time units. Assume the *left laxity* of τ_x^y is 5 and the *right laxity* of τ_x^{y+1} is 5. It implies $s_x^{y+1} - s_x^y \leq 15$. Consider the scheduling of a task instance τ_i^j with execution time 15. If we enlarge the interval between τ_x^y and τ_x^{y+1} by shifting τ_x^y left 5 time units and τ_x^{y+1} right 5 time units, then we get a new interval with 15 time units for τ_i^j. However, it turns out that $s_x^{y+1} = 25$, $s_x^y = 5$, and the relative timing constraints between τ_x^y and τ_x^{y+1} is violated.

5   The Priority-Based Scheduling of a Task Set

We consider the priority-based algorithms for scheduling a set of periodic tasks with hybrid timing constraints. Given a set of periodic tasks $\Gamma = \{ \tau_i \mid i = 1, \ldots, n \}$ with the task characteristics described in Section 2, we compute the LCM of all periods. Each task τ_i is extended to n_i task instances: $\tau_i^1, \tau_i^2, \ldots, \tau_i^{n_i}$. A scheduling algorithm σ for Γ is to totally order the instances of all tasks within the LCM. Namely, σ : *task_id* × *instance_id* → *integer*.

Three algorithms are considered. They are *smallest latest-start-time first* (SLsF), *smallest period first* (SPF), and *smallest jitter first* (SJF) algorithms.

5.1   SLsF

The scheduling window for a task instance τ_i^j depends on the scheduling of its preceding instance. Once s_i^{j-1} is determined, the scheduling window of the instance can be computed by equations 5 and 6. The scheduling window for the first instance of a task τ_i is defined as $[r_i, d_i - e_i]$.

The idea of the SLsF algorithm is to pick one candidate instance with the minimum *lst* among all tasks at a time. One counter for each task is maintained to indicate the candidate instance. All counters are initialized to 1. Each time when a task instance with the smallest *lst* is chosen, the algorithm in Table 3 is invoked to schedule the instance. After the scheduling of the instance is done, the counter is increased by one. The counter for τ_i overflows when it reaches $n_i + 1$. It means

that all the instances of τ_i are scheduled. The algorithm terminates when all counters overflow.

We can compute the relative deadline for a task instance by adding the execution time to the *lst*. If the execution times for all tasks are identical, the SLsF algorithm is equivalent to the *earliest deadline first* (EDF) algorithm.

5.2 SPF

The task periods determine the LCM of Γ and the numbers of instances for tasks within the LCM. In the most cases, the task with the smaller period has the tighter timing constraints. Namely, $(\lambda_i + \eta_i) \leq (\lambda_j + \eta_j)$ if $p_i \leq p_j$. To make the tasks with the smaller periods meet their timing constraints, the SPF algorithm favors the tasks with smaller periods.

The SPF algorithm uses the period as the key to arrange all tasks in non-decreasing order. The task with the smallest period is selected to schedule first. The instances of a particular task are scheduled one by one by invoking the algorithm in Table 3. After all the instances of a task are scheduled, the next task in the sequence is scheduled.

5.3 SJF

We define the *jitter* of a task τ_i as $(\lambda_i + \eta_i)$. It is proportional to the range of the scheduling window. Hence, The schedulability of a task also depends on the jitter.

Instead of using the period as the measurement, the SJF algorithm assigns the higher priority to the tasks with the smaller jitters. The task with the smallest jitter is scheduled first.

5.4 The Solution

The composition of the time-based scheduling of a task instance and the priority assignment of task instances is shown in Figure 5. The priority assignment can be done by using SLsF, SPF, or SJF. The function *Schedule_An_Instance()* is invoked to schedule a single task instance.

6 Experimental Evaluation

We conduct two experiments to study and compare the performance of the three algorithms. The purpose of the first experiment is to study the effect of the number of tasks and utilization on

A set of tasks is given

Find the next unscheduled task instance
By some priority-based assignment,
Such as SLsF, SPF, and SJF.

Schedule_An_Instance() as shown in Table 3

Some instance is unscheduled

All instances are scheduled
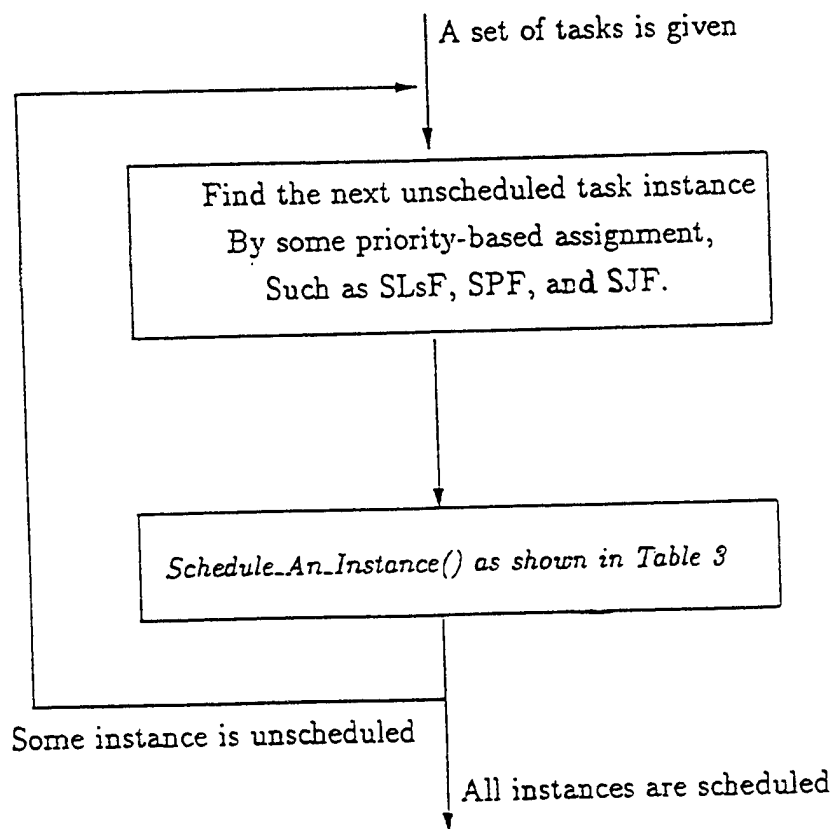
Figure 5: A schematic flowchart for the solution

the schedulability of each algorithm. The objective of the second experiment is to compare the performance of the three algorithms.

6.1 The First Experiment

The task generation scheme for the first experiment is characterized by the following parameters.

- Periods of the tasks: We consider a homogeneous system in which the period of one task could be either the same as or multiple of the period of another. We consider a system with 40, 80, 160, 320, and 640 as the candidate periods. There may be more than one task with the same period.

- The execution time of a task, e_i : It has the uniform distribution over the range $[0, \frac{p_i}{16}]$, where p_i is the period of the task τ_i. The execution time could be a real value.

- The jitters of a task: $\lambda_i = \eta_i = 0.1 \times p_i$.

We define the utilization of a task system as

$$\sum_{i=1}^{N} \frac{e_i}{p_i} \tag{24}$$

In the first experiment, the utilization value and the number of tasks in a set are the controlled variables. Given an utilization value U and the number of tasks N the scheme first generates a run of raw data by randomly generating a set of N tasks based on the the selected periods, jitter values, and the execution time distribution. The utilization of the raw data, u, is then computed by Equation 24. Finally, the utilization value of the raw data is scaled up or down to U by multiplying $\frac{U}{u}$ to the execution time of each generated task. As a consequence, we obtain a set of tasks with the specified (U, N) value.

For each combination of (U, N) in which $U = 5\%, 10\%, 15\%, \ldots 100\%$ and $N = 10, 20,$ and 30, we apply the scheme to generate 5000 cases of input data and use the three algorithms to solve them. The schedulability degree of each (U, N) combination for an algorithm is obtained by dividing the number of solved cases by 5000. Since the jitter values is 1/10 of periods, it is observed that the SPF and SJF algorithms yield the same results. The results are shown in Figure 6.

As can be seen in Figures 6(a) and (b) the number of tasks has the different effects on the three algorithms. For SLsF, given a fixed utilization value, the schedulability degree increases
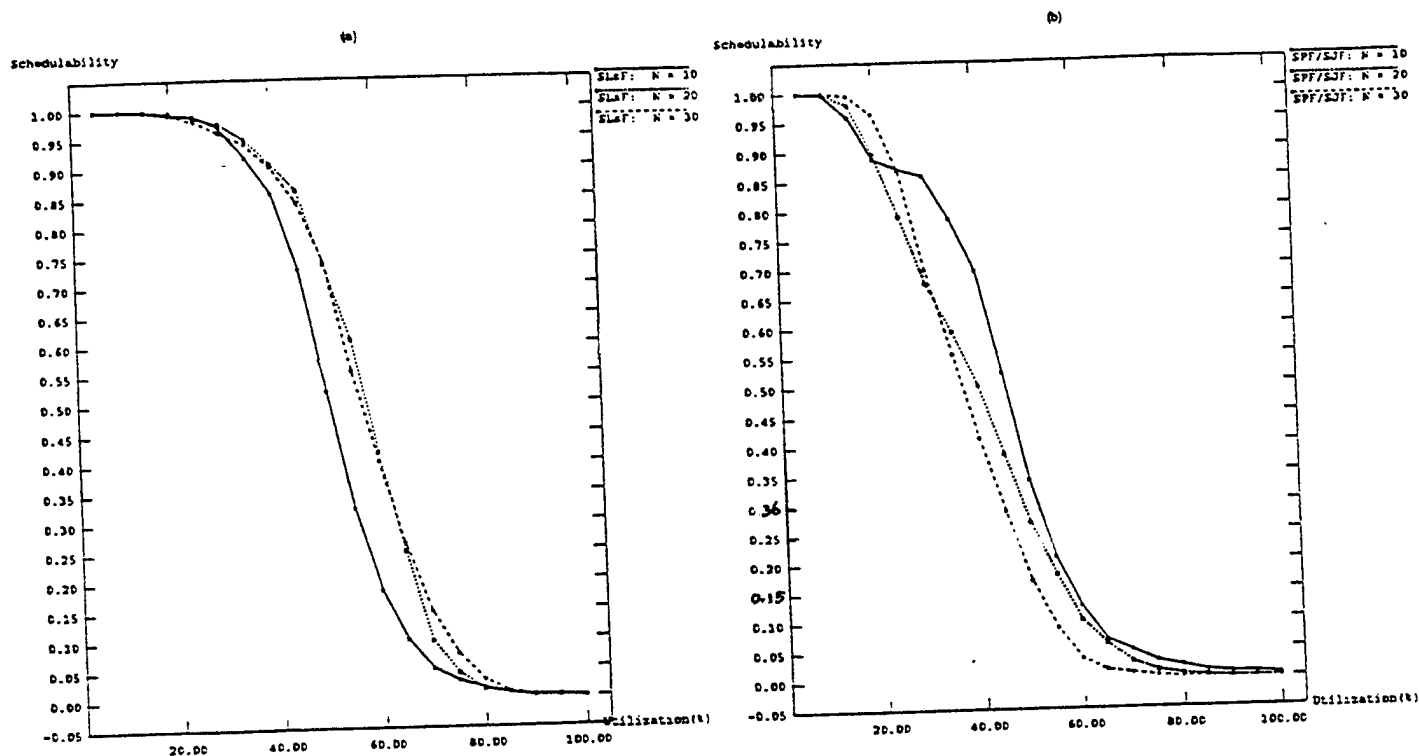
Figure 6: The effect of the numbers of tasks on the schedulability

as the number of tasks in a system becomes bigger. It is beacuse the execution time of a task becomes smaller as the number of tasks increases. For a task system with smaller execution time distribution, the chance for SLsF to find a feasible solution is bigger. The same phenomenon is also found in Figure 6(b) for SPF and SJF in the low-utilization cases (i.e. $U \leq 20\%$). However, for the high-utilization cases in Figure 6(b), the complexity of the number of tasks dominates the algorithms and the schedulability decreases.

6.2 The Second Experiment

The task generation scheme for the second experiment is characterized by the following parameters.

- LCM = 300

- The number of tasks is 20.

- Periods of the tasks: We consider the factors of the LCM as the periods. They are 20, 30, 50, 60, 100, 150, and 300. There may be more than one task with the same period.

145

- The execution time of a task, e_i : It has the uniform distribution over the range $[0, \frac{p_i}{15}]$, where p_i is the period of the task τ_i. The execution time could be a real value.

- The jitters of a task: $\lambda_i = \eta_i = 0.1 \times p_i + 2 \times e_i$.

The generation scheme for the second experiment is similar to the first one. Given an utilization value U, a set of 20 tasks is randomly generated according to the parameters listed above and then the execution time of each task is normalized in order to make the utilization value equal to U exactly.

We generate 5000 cases of different task sets for each utilization value ranging from 0.05 to 1.00. The schedulability degree of each algorithm on a particular utilization value is obtained by dividing the number of solved cases by 5000. We compare the schedulability degrees of the algorithms on different utilization values. The results are shown in Figure 7(a).

As can be see in Figure 7(a) the SLsF algorithm outperforms the other two algorithms. For example, when the utilization = 50%, the schedulability degree of SLsF is 0.575 while those of SPF and SJF are less than 0.2. It is because the way of assigning the priorities to the task instances in the SLsF algorithm reflects the urgency of task instances by considering the latest start times.

We also compare the objective function value π in Equation 21 among the three algorithms. We define the normalized objective function for an algorithm as

$$\bar{z} = \sum_{i=1}^{5000} \frac{z_i}{5000},$$ (25)

where $z_i = \begin{cases} 1 & \text{if the algorithm can not find a feasible solution to case } i. \\ 0 & \text{if } max(i) = min(i). \\ \frac{\pi_i - min(i)}{max(i) - min(i)} & \text{otherwise.} \end{cases}$

Given case i, the values of $min(i)$ and $max(i)$ are calculated among the objective values obtained from the algorithms which solve the case. For the algorithms which can not find a feasible solution to case i, the objective values are not taken into account when $min(i)$ and $max(i)$ are calculated. The results of the normalized objective functions for each algorithm on different utilization values are shown in Figure 7(b).

It is observed that in the low-utilization cases SJF finds feasible solutions with smaller objective values. It is because that SJF schedules the tasks with the smallest jitters first. By scheduling the tasks with smaller jitter value first it is more easier to make the instances of a task one period apart. we can find a feasible solution with smaller objective value. However, in the middle- or
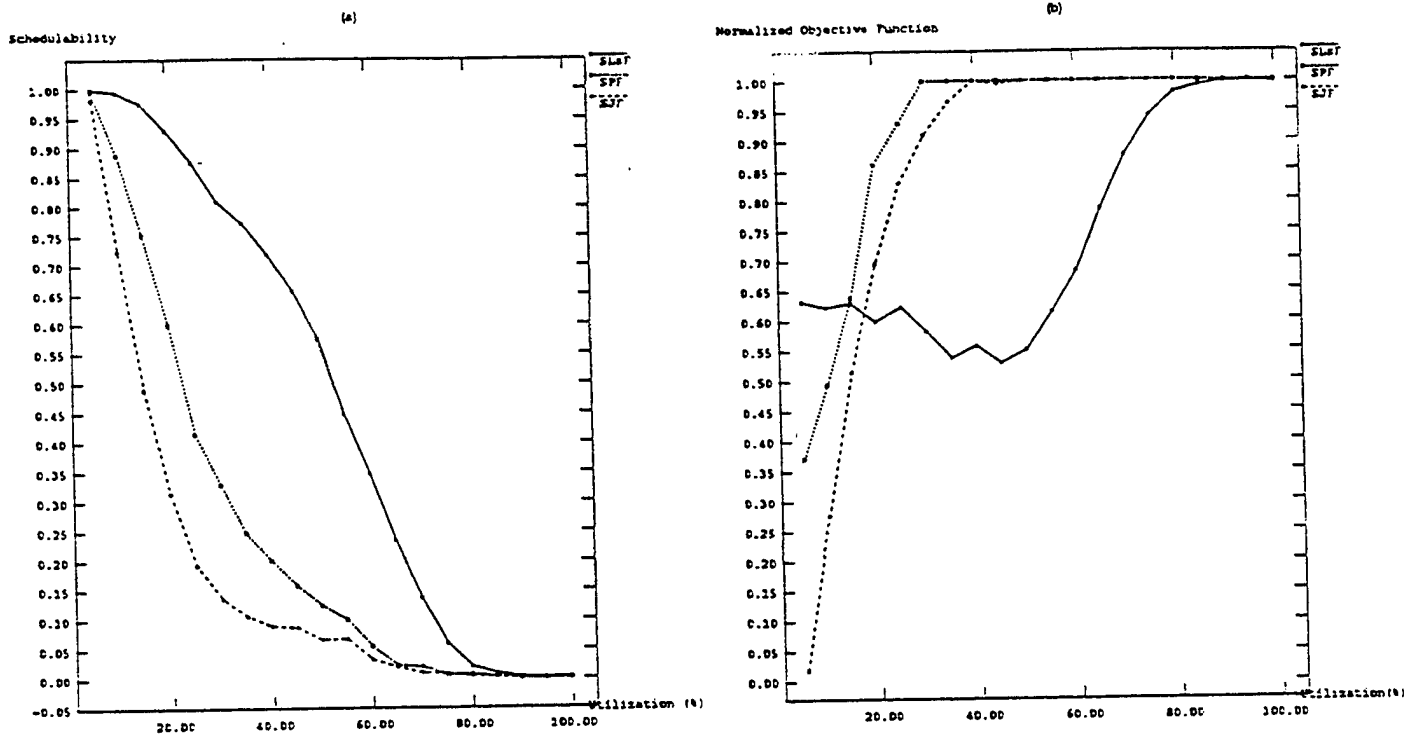
Figure 7: The comparison of three algorithms

high-utilization cases, the schedulability dominates the normalized objective function, and SLsF outperforms the other two algorithms in these regions.

7 Summary

In this paper we have considered the static non-preemptive scheduling algorithm on a single processor for a set of periodic tasks with hybrid timing constraints. The time-based scheduling algorithm is used to schedule a task instance once the scheduling window of the instance is given. We also have presented three priority assignment algorithms for the task instances and conducted experiments to compare the performance. From the experimental results, we see that the SLsF outperforms the other two algorithms.

The techniques presented in this chapter can be applied to multi-processor real-time systems. Communication and synchronization constraints can be also incorporated. In our future work, the extension to a distributed computing systems will be investigated.

References

[1] T. Carpenter, K. Driscoll, K. Hoyme, and J. Carciofini. Arinc 659 scheduling: Problem definition. In *Proceedings of IEEE Real-Time Systems Symposium*, San Juan, PR, Dec. 1994.

[2] M. L. Dertouzos and A.K. Mok. Multiprocessor on-line scheduling of hard real-time tasks. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, Dec. 1989.

[3] M.G. Harbour, M.H. Klein, and J.P. Lehoczky. Fixed priority scheduling of periodic tasks with varying execution priority. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 116–128, Dec. 1991.

[4] Krithi Ramamritham. Allocation and scheduling of complex periodic tasks. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 108–115, Paris, France, 1990.

[5] T. Shepard and J.A.M. Gagne. A pre-run-time scheduling algorithm for hard real-time systems. *IEEE Transactions on Software Engineering*, 17(7):669–677, July 1991.

[6] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2), March 1994.

[7] J.P.C. Verhoosel, E.J. Luit, D.K. Hammer, and E. Jansen. A static scheduling algorithm for distributed hard real-time systems. *Real-Time Systems*, pages 227–246, 1991.

[8] J. Xu and D.L. Parnas. Scheduling processes with release times, deadlines, precedence, and exclusion relations. *IEEE Transactions on Software Engineering*, 16(3):360–369, March 1990.

Schedule_An_Instance (τ_i^j):

Input: A linked list, a task instance τ_i^j and a sequence of sorted free intervals, $S = \{\ int_1,\ int_2,\ \ldots,\ int_{|S|}\ \}$, in which each interval overlaps $[est(\tau_i^j), lst(\tau_i^j)]$.

Let the execution time of τ_i^j be e.

For $n = 1$ to $|S|$ do

      Let int_n be $[\ell, \mu]$.

      If $\mu - \ell \geq e$ then

            Return a new time slot with *start time* $= \ell$ and *finish time* $= \ell + e$.

      End if.

End for.

Compute *left laxity* and *right laxity* for each time slot in the linked list by equations 22 and 23.

For $n = 1$ to $|S|$ do

      Let int_n be $[\ell, \mu]$.

      If $\ell \geq s_i^{j-1} + p_i$ then /* Try left shift first then right shift */

            Let the time slot that immediately precedes int_n be w_k.

            If *left_laxity*$(w_k) + \mu - \ell \geq e$ then /* Left shift */

                Left_Shift$(w_k, e - \mu + \ell)$.

                Return a new time slot with *start time* $= \mu - e$ and *finish time* $= \mu$.

            Else

                Let the time slot that immediately follows int_n be w_k.

                If *right_laxity*$(w_k) + \mu - \ell \geq e$ then /* Right shift */

                    Right_Shift$(w_k, e - \mu + \ell)$.

                    Return a new time slot with *start time* $= \ell$ and *finish time* $= \ell + e$.

                End If.

            End If.

      Else /* Try right shift first then left shift */

            Let the time slot that immediately follows int_n be w_k.

            If *right_laxity*$(w_k) + \mu - \ell \geq e$ then /* Right shift */

                Right_Shift$(w_k, e - \mu + \ell)$.

                Return a new time slot with *start time* $= \ell$ and *finish time* $= \ell + e$.

            Else

                Let the time slot that immediately precedes int_n be w_k.

                If *left_laxity*$(w_k) + \mu - \ell \geq e$ then /* Left shift */

                    Left_Shift$(w_k, e - \mu + \ell)$.

                    Return a new time slot with *start time* $= \mu - e$ and *finish time* $= \mu$.

                End If.

            End If.

      End If.

End for.

Schedule τ_i^j at the end of linked list.

Table 3: The Scheduling of a Task Instance

REPORT DOCUMENTATION PAGE

Form approved
OMB No 074-0188

1. AGENCY USE ONLY (leave blank)	2. REPORT DATE 1/3/1995	3. REPORT TYPE END DATES COVERED Technical Reports

4. TITLE AND SUBTITLE
Scheduling of Periodic Tasks with Relative Timing Constraints

5. FUNDING NUMNBERS
N00014-91-C-0195

6. AUTHOR(S)
S.-T. Cheng and A. K. Agrawala

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Maryland
Department of Computer Science
A.V. Williams Building
College Park, MD 20742

8. PERFORMING ORGANIZATION REPORT NUMBER
CSTR 3392

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)
PhillipsLaboratory
Director of Contracting
3651 Lowry Avenue SE
Kirtland AFB, NM 87117-5777

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12.a. DISTRIBUTION/ AVAILABILITY STATEMENT

12.b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
The problem of non-preemptive scheduling of a set of periodic tasks on a single processor has been traditionally considering the ready time and deadline on each task. As a consequence, a feasible schedule finds that in each period one instance of each task starts the execution after the ready time and completes the execution before the deadline. Recently, the timing requirements of the real-time systems emerge that the relative timing constraints are imposed on the consecutive executions of each task. In this paper, we consider the scheduling problem of the periodic tasks with the relative timing constraints imposed on two consecutive executions of a task. We analyze the timing constraints and derive the scheduling window for each task instance. Based on the scheduling window, we present the time-based approach of scheduling a task instance. The task instances are scheduled one by one based on their priorities assigned by the proposed algorithms in this paper. We conduct the experiments to compare the schedulability of the algorithms.

14. SUBJECT TERMS
Time constraints, timing requirements

15. NUMBER OF PAGES
21

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT Umlimited

A Scalable Virtual Circuit Routing Scheme for ATM Networks*

Cengiz Alaettinoğlu
Information Sciences Institute
University of Southern California
Marina del Rey, CA 90292

Ibrahim Matta   A. Udaya Shankar
Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
College Park, MD 20742

October 1994

Abstract

High-speed networks, such as ATM networks, are expected to support diverse quality-of-service (QoS) requirements, including real-time QoS. Real-time QoS is required by many applications such as voice and video. To support such service, routing protocols based on the Virtual Circuit (VC) model have been proposed. However, these protocols do not scale well to large networks in terms of storage and communication overhead.

In this paper, we present a scalable VC routing protocol. It is based on the recently proposed viewserver hierarchy, where each viewserver maintains a partial view of the network. By querying these viewservers, a source can obtain a merged view that contains a path to the destination. The source then sends a request packet over this path to setup a real-time VC through resource reservations. The request is blocked if the setup fails. We compare our protocol to a simple approach using simulation. Under this simple approach, a source maintains a full view of the network. In addition to the savings in storage, our results indicate that our protocol performs close to or better than the simple approach in terms of VC carried load and blocking probability over a wide range of real-time workload.

Categories and Subject Descriptors: C.2.1 [Computer-Communication Networks]: Network Architecture and Design—*packet networks; store and forward networks*; C.2.2 [Computer-Communication Networks]: Network Protocols—*protocol architecture*; C.2.m [Routing Protocols]; F.2.m [Computer Network Routing Protocols].

151

Contents

1   Introduction

Integrated services packet-switched networks, such as Asynchronous Transfer Mode (ATM) networks [21], are expected to carry a wide variety of applications with heterogeneous quality of service (QoS) requirements. For this purpose, new resource allocation algorithms and protocols have been proposed, including link scheduling, admission control, and routing. Link scheduling defines how the link bandwidth is allocated among the different services. Admission control defines the criteria the network uses to decide whether to accept or reject a new incoming application. Routing concerns the selection of routes to be taken by application packets (or cells) to reach their destination. In this paper, we are mainly concerned with routing for real-time applications (e.g., voice, video) requiring QoS guarantees (e.g., bandwidth and delay guarantees).

To provide real-time QoS support, a number of *virtual-circuit* (VC) routing approaches have been proposed. A *simple* (or straightforward) approach to VC routing is the link-state full-view approach. Here, each end-system maintains a view of the whole network, i.e. a graph with a vertex for every node[1] and an edge between two neighbor nodes. QoS information such as delay, bandwidth, and loss rate are attached to the vertices and the edges of the view. This QoS information is flooded regularly to all end-systems to update their views. When a new application requests service from the network, the source end-system uses its current view to select a *source route* to the destination end-system that is likely to support the application's requested QoS, i.e., a sequence of node ids starting from the source end-system and ending with the destination end-system. A VC-setup message is then sent over the selected source route to try to reserve the necessary resources (bandwidth, buffer space, service priority) and establish a VC.

Typically, at every node the VC-setup message visits, a set of admission control tests are performed to decide whether the new VC, if established, can be guaranteed its requested QoS without violating the QoS guaranteed to already established VCs. At any node, if these admission tests are passed, then resources are reserved and the VC-setup message is forwarded to the next node. On the other hand, if the admission tests fail, a VC-rejected message is sent back towards the source node releasing resource reservations made by the VC-setup message, and the application request is either blocked or another source route is selected and tried. If the final admission tests at the destination node are passed, then a VC-established message is sent back towards the source node confirming resource reservations made during the forward trip of the VC-setup message. Upon receiving the VC-established message, the application can start transmitting its packets over its

---

[1] We refer to switches and end-systems collectively as nodes.

153

reserved VC. This VC is torn down and resources are released at the end of the transmission.

Clearly, the above simple routing scheme does not scale up to large networks. The storage at each end-system and the communication cost are proportional to $N \times d$, where N is the number of nodes and d is the average number of neighbors to a node.

A traditional solution to this scaling problem is the area hierarchy used in routing protocols such as the Open Shortest Path First (OSPF) protocol [18]. The basic idea is to aggregate nodes hierarchically into areas: "close" nodes are aggregated into level 1 areas, "close" level 1 areas are aggregated into level 2 areas, and so on. An end-system maintains a view that contains the nodes in the same level 1 area, the level 1 areas in the same level 2 area, and so on. Thus an end-system maintains a smaller view than it would in the absence of hierarchy. Each area has its own QoS information derived from that of the subareas. A major problem of an area-based scheme is that aggregation results in loosing detailed link-level QoS information. This decreases the chance of the routing algorithm to choose "good" routes, i.e. routes that result in high successful VC setup rate (or equivalently high carried VC load).

Our scheme

In this paper, we present a scalable VC routing scheme that does not suffer from the problems of areas. Our scheme is based on the viewserver hierarchy we recently proposed in [3, 2] for large internetworks and evaluated for administrative policy constraints. Here, we are concerned with the support of performance/QoS requirements in large wide-area ATM-like networks, and we adapt our viewserver protocols accordingly.

In our scheme, views are not maintained by every end-system but by special switches called *viewservers*. For each viewserver, there is a subset of nodes around it, referred to as the viewserver's *precinct*. The viewserver only maintains the view of its precinct. This solves the scaling problem for storage requirement.

A viewserver can provide source routes for VCs between source and destination end-systems in its precinct. Obtaining a route between a source and a destination that are not in any single view involves accumulating the views of a sequence of viewservers. To make this process efficient, viewservers are organized hierarchically in levels, and an associated addressing structure is used. Each end-system has a set of addresses. Each *address* is a sequence of viewserver ids of decreasing levels, starting at the top level and going towards the end-system. The idea is that when the views of the viewservers in an address are merged, the merged view contains routes to the end-system

154

from the top level viewservers.

We handle dynamic topology changes such as node/link failures and repairs, and link cost changes. Nodes detect topology changes affecting itself and neighbor nodes. Each node communicates these changes by flooding to the viewservers in a specified subset of nodes; this subset is referred to as its *flood area*. Hence, the number of packets used during flooding is proportional to the size of the flood area. This solves the scaling problem for the communication requirement.

Thus our VC routing protocol consists of two subprotocols: a *view-query protocol* between end-systems and viewservers for obtaining merged views; and a *view-update protocol* between nodes and viewservers for updating views.

Evaluation

In this paper, we compare our viewserver-based VC routing scheme to the simple scheme using VC-level simulation. In our simulation model, we define network topologies, QoS requirements, viewserver hierarchies, and evaluation measures. To the best of our knowledge, this is the first evaluation of a dynamic hierarchical-based VC routing scheme under real-time workload.

Our evaluation measures are the amount of memory required at the end-systems, the amount of time needed to construct a path[2], the carried VC load, and the VC blocking probability. We use network topologies each of size 2764 nodes. Our results indicate that our viewserver-based VC routing scheme performs close to or better than the simple scheme in terms of VC carried load and blocking probability over a wide range of workload. It also reduces the amount of memory requirement by up to two order of magnitude.

Organization of the paper

In Section 2, we survey recent approaches to VC routing. In Section 3, we present the view-query protocol for static network conditions, that is, assuming all links and nodes of the network remain operational. In Section 4, we present the view-update protocol to handle topology changes. In Section 5, we present our evaluation model. Our results are presented in Section 6. Section 7 concludes the paper.

---

[2] We use the terms route and path interchangeably.

2 Related Work

In this section, we discuss routing protocols recently proposed for packet-switched QoS networks. These routing protocols can be classified depending on whether they help the network support *qualitative* QoS or *quantitative* (real-time) QoS. For a qualitative QoS, the network tries to provide the service requested by the application with no performance guarantees. Such a service is often identified as "best-effort". A quantitative QoS provides performance guarantees (typically required by real-time applications); for example, an upper bound on the end-to-end delay for any packet received at the destination.

Routing protocols that make routing decisions on a per VC basis can be used to provide either qualitative or quantitative QoS. For a quantitative QoS, some admission control tests should be performed during the VC-setup message's trip to the destination to try to reserve resources along the VC's path as described in Section 1.

On the other hand, the use of routing protocols that make routing decisions on a per packet basis is problematic in providing resource guarantees [5], and qualitative QoS is the best service the network can offer.

Since we are concerned in this paper with real-time QoS, we limit our following discussion to VC routing schemes proposed or evaluated in this context. We refer the reader to [19, 6] for a good survey on many other routing schemes.

Most of the VC routing schemes proposed for real-time QoS networks are based on the link-state full-view approach described in Section 1 [6, 1, 10, 24]. Recall that in this approach, each end-system maintains a view of the whole network, i.e. a graph with a vertex for every node and an edge between two neighbor nodes. QoS information is attached to the vertices and the edges of the view. This QoS information is distributed regularly to all end-systems to update their views and thus enable the selection of appropriate source routes for VCs, i.e. routes that are likely to meet the requested QoS. The proposed schemes mainly differ in how this QoS information is used. Generally, a cost function is defined in terms of the QoS information, and used to estimate the cost of a path to the VC's destination. The route selection algorithm then favors short paths with minimum cost. See [17, 22] for an evaluation of several schemes.

A number of VC routing schemes have also been designed for networks using the Virtual Path (VP) concept [15, 14]. This VP concept has been proposed to simplify network management and control by having separate (logically) fully-connected subnetworks, typically one for each service class. In each VP subnetwork, simple routing schemes that only consider one-hop and two-hop

paths are used. However, the advantage of using VPs can be offset by a decrease in statistical multiplexing gains of the subnetworks [15]. In this work, we are interested in *general* network topologies, where the shortest paths can be of arbitrary hop length and the overhead of routing protocols is of much concern.

All the above VC routing schemes are based on the link-state approach. VC routing schemes based on the path-vector approach have also been proposed [13]. In this approach, for each destination a node maintains a set of paths, one through each of its neighbor nodes. QoS information is attached to these paths. For each destination, a node exchanges its best feasible path[3] with its neighbor nodes. The scheme in [13] provides two kinds of routes: pre-computed and on-demand. Pre-computed routes match some well-known QoS requirements, and are maintained using the path-vector approach. On-demand routes are calculated for specific QoS requirements upon request. In this calculation, the source broadcasts a special packet over all candidate paths. The destination then selects a feasible path from them and informs the source [13, 23]. One drawback of this scheme is that obtaining on-demand routes is very expensive since there are potentially exponential number of candidate paths between the source and the destination.

The link-state approach is often proposed and favored over the path-vector approach in QoS architectures for several reasons [16]. An obvious reason is simplicity and complete control of the source over QoS route selection.

The above VC routing schemes do not scale well to large QoS networks in terms of storage and communication requirements. Several techniques to achieve scaling exist. The most common technique is the area hierarchy described in Section 1.

The landmark hierarchy [26, 25] is another approach for solving the scaling problem. The link-state approach can not be used with the landmark hierarchy. A thorough study of enforcing QoS and policy constraints with this hierarchy has not been done.

Finally, we should point out that extensive effort is currently underway to fully specify and standardize VC routing schemes for the future integrated services Internet and ATM networks [9].

3   Viewserver Hierarchy Query Protocol

In this section, we present our scheme for static network conditions, that is, all links and nodes remain operational. The dynamic case is presented in Section 4.

---

[3] A feasible path is a path that satisfies the QoS constraints of the nodes in the path.

Conventions: Each node has a unique id. NodeIds denotes the set of node-ids. For a node u, we use $nodeid(u)$ to denote the id of u. $NodeNeighbors(u)$ denotes the set of ids of the neighbors of u.

In our protocol, a node u uses two kinds of sends. The first kind has the form "Send(m) to v", where m is the message being sent and v is the destination-id. Here, nodes u and v are neighbors, and the message is sent over the physical link (u, v). If the link is down, we assume that the packet is dropped.

The second kind of send has the form "Send(m) to v using sr", where m and v are as above and sr is a source route between u and v. We assume that as long as there is a sequence of up links connecting the nodes in sr, the message is delivered to v. This requires a transport protocol support such as TCP [20].

To implement both kind of sends, we assume there is a reserved VC on each link for sending routing, signaling and control messages [4]. This also ensures that routing messages do not degrade the QoS seen by applications.

Views and Viewservers

Views are maintained by special nodes called *viewservers*. Each viewserver has a *precinct*, which is a set of nodes around the viewserver. A viewserver maintains a *view*, consisting of the nodes in its precinct, links between these nodes and links outgoing from the precinct[4]. Formally, a viewserver x maintains the following:

$Precinct_x \subseteq$ NodeIds. Nodes whose view is maintained.

$View_x$. View of x.
$$= \{\langle u, \ timestamp, \ expirytime, \ \{\langle v, \ cost \rangle : \ v \in NodeNeighbors(u)\} \rangle :$$
$$u \in Precinct_x\}$$

The intention of $View_x$ is to obtain source routes between nodes in $Precinct_x$. Hence, the choice of nodes to include in $Precinct_x$ and the choice of links to include in $View_x$ are not arbitrary. $Precinct_x$ and $View_x$ must be connected; that is, between any two nodes in $Precinct_x$, there should be a path in $View_x$. Note that $View_x$ can contain links to nodes outside $Precinct_x$. We say that a node u is *in the view* of a viewserver x, if either u is in the precinct of x, or $View_x$ has a link from a node in the precinct of x to node u. Note that the precincts and views of different viewservers can be overlapping, identical or disjoint.

---

[4] Not all the links need to be included.

158

For a link (u, v) in the view of a viewserver x, $View_x$ stores a *cost*. The cost of the link (u, v) equals a vector of values if the link is known to be up; each cost value estimates how expensive it is to cross the link according to some QoS criteria such as delay, throughput, loss rate, etc. The cost equals ∞ if the link is known to be down. Cost of a link changes with time (see Section 4). The view also includes *timestamp* and *expirytime* fields which are described in Section 4.

Viewserver Hierarchy

For scaling reasons, we cannot have one large view. Thus, obtaining a source route between a source and a destination which are far away, involves accumulating views of a sequence of viewservers. To keep this process efficient, we organize viewservers hierarchically. More precisely, each viewserver is assigned a hierarchy level from $0, 1, \ldots$, with 0 being the top level in the hierarchy. A parent-child relationship between viewservers is defined as follows:

1. Every level i viewserver, $i > 0$, has a parent viewserver whose level is less than i.

2. If viewserver x is a parent of viewserver y then x's precinct contains y and y's precinct contains x.

3. The precinct of a top level viewserver contains all other top level viewservers.

In the hierarchy, a parent can have many children and a child can have many parents. We extend the range of the parent-child relationship to ordinary nodes; that is, if $Precinct_x$ contains the node u, we say that u is a child of x, and x is a parent of u. We assume that there is at least one parent viewserver for each node.

For a node u, an address is defined to be a sequence $\langle x_0, x_1, \ldots, x_t \rangle$ such that x_i for $i < t$ is a viewserver-id, x_0 is a top level viewserver-id, x_t is the id of u, and x_i is a parent of x_{i+1}. A node may have many addresses since the parent-child relationship is many-to-many. If a source node wants to establish a VC to a destination node, it first queries the name servers to obtain a set of addresses for the destination[5]. Second, it queries viewservers to obtain an accumulated view containing both itself and the destination node (it can reach its parent viewservers by using fixed source routes to them). Then, it chooses a feasible source route from this accumulated view and initiates the VC setup protocol on this path.

View-Query Protocol: Obtaining Source Routes

We now describe how a source route is obtained.

---

[5] Querying the name servers can be done in the same way as is done currently in the Internet.

We want a sequence of viewservers whose merged views contains both the source and the destination nodes. Addresses provide a way to obtain such a sequence, by first going up in the viewserver hierarchy starting from the source node and then going down in the viewserver hierarchy towards the destination node. More precisely, let $\langle s_0, \ldots, s_t \rangle$ be an address of the source. and $\langle d_0, \ldots, d_l \rangle$ be an address of the destination. Then, the sequence $\langle s_{t-1}, \ldots, s_0, d_0, \ldots, d_{l-1} \rangle$ meets our requirements. In fact, going up all the way in the hierarchy to top level viewservers may not be necessary. We can stop going up at a viewserver s_i if there is a viewserver $d_j, j < l$, in the view of s_i (one special case is where $s_i = d_j$).

The view-query protocol uses two message types:

- (RequestView, *s_address*, *d_address*)

  where *s_address* and *d_address* are the addresses for the source and the destination respectively. A RequestView message is sent by a source node to obtain an accumulated view containing both the source and the destination nodes. When a viewserver receives a RequestView message, it either sends back its view or forwards this request to another viewserver.

- (ReplyView, *s_address*, *d_address*, *accumview*)

  where *s_address* and *d_address* are as above and *accumview* is the accumulated view. A ReplyView message is sent by a viewserver to the source or to another viewserver closer to the source. The *accumview* field in a ReplyView message equals the union of the views of the viewservers the message has visited.

We now describe the view-query protocol in more detail (please refer to Figures 1 and 2). To establish a VC to a destination node, the source node sends a RequestView packet containing the source and the destination addresses to its parent in the source address.

Upon receiving a RequestView packet, a viewserver x checks if the destination node is in its precinct[6]. If it is, x sends back its view in a ReplyView packet. If it is not, x forwards the request packet to another viewserver as follows (details in Figure 2): x checks whether any viewserver in the destination address is in its view. If there is such a viewserver, x sends the RequestView packet to the last such one in the destination address. Otherwise x is a viewserver in the source address, and it sends the packet to its parent in the source address.

When a viewserver x receives a ReplyView packet, it merges its view to the accumulated view in the packet. Then it sends the ReplyView packet towards the source node in the same way it would send a RequestView packet towards the destination node (i.e. the roles of the source address

---

[6] Even though the destination can be in the view of x, its QoS characteristics is not in the view if it is not in the precinct of x.

160

```
Constants

  FixedRoutes_u(x), for every viewserver-id x such that x is a parent of u,
       = {⟨y_1, ..., y_n⟩ : y_i ∈ NodeIds}. Set of routes to x

Events
  RequestView_u(s_address, d_address)          {Executed when u wants a source route}
      Let s_address be ⟨s_0, ..., s_{t-1}, s_t⟩, and sr ∈ FixedRoutes_u(s_{t-1});
      Send(RequestView, s_address, d_address) to s_{t-1} using sr

  Receive_u(ReplyView, s_address, d_address, accumview)
      Choose a feasible source route using accumview;
      If a feasible route is not found
            Execute RequestView_u again with another source address and/or destination address
```

Figure 1: View-query protocol: Events and state of a source node u.

```
Constants

  Precinct_x. Precinct of x.

Variables

  View_x. View of x.

Events
  Receive_x(RequestView, s_address, d_address)
      Let d_address be ⟨d_0, ..., d_t⟩;
      if d_t ∉ Precinct_x then
      forward_x(RequestView, s_address, d_address, {});
      else forward_x(ReplyView, d_address, s_address, View_x);        {addresses are switched}
      endif

  Receive_x(ReplyView, s_address, d_address, view)
      forward_x(ReplyView, s_address, d_address, view ∪ View_x)

  where procedure forward_x(type, s_address, d_address, view)
      Let s_address be ⟨s_0, ..., s_t⟩, d_address be ⟨d_0, ..., d_l⟩;
      if ∃i : d_i in View_x then
      Let i = max{j : d_j in View_x};
      target := d_i;
      else target := s_i such that s_{i+1} = nodeid(x);
      endif;
      sr := choose a route to target from nodeid(x) using View_x;
      if type = RequestView then
      Send(RequestView, s_address, d_address) to target using sr;
      else Send(ReplyView, s_address, d_address, view) to target using sr;
      endif
```

Figure 2: View-query protocol: Events and state of a viewserver x.

and the destination address are interchanged).

When the source receives a ReplyView packet, it chooses a feasible path using the *accumview* in the packet. If it does not find a feasible path, it can try again using a different source and/or destination addresses. Note that the source does not have to throw away the previous accumulated views; it can merge them all into a richer accumulated view. In fact, it is easy to change the protocol so that the source can also obtain views of individual viewservers to make the accumulated view even richer. Once a feasible source route is found, the source node initiates the VC setup protocol.

Above we have described one possible way of obtaining the accumulated views. There are various other possibilities, for example: (1) restricting the ReplyView packet to take the reverse of the path that the RequestView packet took; (2) having ReplyView packets go all the way up in the viewserver-hierarchy for a richer accumulated view; (3) having the source poll the viewservers directly instead of the viewservers forwarding request/reply messages to each other; (4) not including non-transit nodes (e.g. end-systems) other than the source and the destination nodes in the *accumview*; (5) including some QoS requirements in the RequestView packet, and having the viewservers filter out some nodes and links.

4   Update Protocol for Dynamic Network Conditions

In this section, we first describe how topology changes such as link/node failures, repairs and cost changes, are detected and communicated to viewservers, i.e. the view-update protocol. Then, we modify the view-query protocol appropriately.

View-Update Protocol: Updating Views

Viewservers do not communicate with each other to maintain their views. Nodes detect and communicate topology changes to viewservers. Updates are done periodically and also optionally after a change in the outgoing link costs.

The communication between a node and viewservers is done by flooding over a set of nodes. This set is referred to as the *flood area*. The topology of a flood area must be a connected graph. For efficiency, the flood area can be implemented by a hop-count.

Due to the nature of flooding, a viewserver can receive information out of order from a node. In order to avoid old information replacing new information, each node includes successively increasing time stamps in the messages it sends. The *timestamp* field in the view of a viewserver equals the largest timestamp received from each node.

Due to node and link failures, communication between a node and a viewserver can fail, resulting in the viewserver having out-of-date information. To eliminate such information, a viewserver deletes any information about a node if it is older than a *time-to-die* period. The *expirytime* field in the view of a viewserver equals the end of the time-to-die period for a node. We assume that nodes send messages more often than the *time-to-die* value (to avoid false removal).

The view-update protocol uses one type of message as follows:

- (Update, *nid*, *timestamp*, *floodarea*, *ncostset*)

  is sent by the node to inform the viewservers about current costs of its outgoing links. Here, *nid* and *timestamp* indicate the id and the time stamp of the node, *ncostset* contains a cost for each outgoing link of the node, and *floodarea* is the set of nodes that this message is to be sent over.

Constants:

  $FloodArea_g$. ($\subseteq$ NodeIds). The flood area of the node.

Variables:

  $Clock_g$: Integer. Clock of g.

Figure 3: State of a node g.

The state maintained by a node g is listed in Figure 3. We assume that consecutive reads of $Clock_g$ returns increasing values.

Constants:

  $Precinct_x$. Precinct of x.

  $TimeToDie_x$: Integer. Time-to-die value.

Variables:

  $View_x$. View of x.

  $Clock_x$: Integer. Clock of x.

Figure 4: State of a viewserver x.

The state maintained by a viewserver x is listed in Figure 4.

The events of node g are specified in Figure 5. The events of a viewserver x are specified in Figure 6. When a viewserver x recovers, $View_x$ is set to $\{\}$. Its view becomes up-to-date as it receives new information from nodes (and remove false information with the time-to-die period).

```
Update_g        {Executed periodically and also optionally upon a change in outgoing link costs}
    ncostset := compute costs for each outgoing link;
    flood_g((Update, nodeid(g), Clock_g, FloodArea_g, ncostset));

Receive_g(packet)      {an Update packet}
    flood_g(packet)

where procedure flood_g(packet)
    if nodeid(g) ∈ packet.floodarea then
    {remove g from the flood area to avoid infinite exchange of the same message.}
    packet.floodarea := packet.floodarea — {nodeid(g)};
    for all h ∈ NodeNeighbors(g) ∧ h ∈ packet.floodarea do
    Send(packet) to h;
    endif


Node Failure Model:   A node can undergo failures and recoveries at anytime. We assume failures are
fail-stop (i.e. a failed node does not send erroneous messages).
```

Figure 5: View-update protocol: Events of a node g.

```
Receive_x(Update, nid, ts, FloodArea, ncset)
    if nid ∈ Precinct_x then
    if ∃(nid, timestamp, expirytime, ncostset) ∈ View_x ∧ ts > timestamp then
    {received is more recent;  delete the old one}
    delete (nid, timestamp, expirytime, ncostset) from View_x;
    endif
    if ¬∃(nid, timestamp, expirytime, ncostset) ∈ View_x then
    ncostset := subset of edge-cost pairs in ncset that are in View_x;
    insert (nid, ts, Clock_x + TimeToDie_x, ncostset) to View_x;
    endif
    endif

Delete_x        {Executed periodically to delete entries older than the time-to-die period}
    for all (nid, tstamp, expirytime, ncset) ∈ View_x ∧ expirytime < Clock_x do
    delete (nid, tstamp, expirytime, ncset) from View_x;

Viewserver Failure Model:   A viewserver can undergo failures and recoveries at anytime. We assume
failures are fail-stop. When a viewserver x recovers, View_x is set to {}.
```

Figure 6: View update events of a viewserver x.

Changes to View-Query Protocol

We now enumerate the changes needed to adapt the view-query protocol to the dynamic case (the formal specification is omitted for space reasons).

Due to link and node failures, RequestView and ReplyView packets can get lost. Hence, the

source may never receive a ReplyView packet after it initiates a request. Thus, the source should try again after a time-out period.

When a viewserver receives a RequestView message, it should reply with its views only if the destination node is in its precinct and its view contains a path to the destination. Similarly during forwarding of RequestView and ReplyView packets, a viewserver, when checking whether a node is in its view, should also check if its view contains a path to it.

5   Evaluation

In this section, we present the parameters of our simulation model. We use this model to compare our viewserver-based VC routing protocols to the simple approach. The results obtained are presented in Section 6.

Network Parameters

We model a campus network which consists of a campus backbone subnetwork and several department subnetworks. The backbone network consists of backbone switches and backbone links.

Each department network consists of a hub switch and several non-hub switches. Each non-hub switch has a link to the department's hub switch. And the department's hub switch has a link to one of the backbone switches. A non-hub switch can have links to other non-hub switches in the same department, to non-hub switches in other departments, or to backbone switches.

End-systems are connected to non-hub switches. An example network topology is shown in Figure 7.

In our topology, there are 8 backbone switches and 32 backbone links. There are 16 departments. There is one hub-switch in each department. There is a total of 240 non-hub switches randomly assigned to different departments. There are 2500 end-systems which are randomly connected to non-hub switches. Thus, we have a total of 2764 nodes.

In addition to the links connecting non-hub switches to the hub switches and hub switches to the backbone switches, there are 720 links from non-hub switches to non-hub switches in the same department, there are 128 links from non-hub switches to non-hub switches in different departments, and there are 64 links from non-hub switches to backbone switches.

The end-points of each link are chosen randomly. However, we make sure that the backbone network is connected; and there is a link from node u to node v iff there is a link from node v to

165
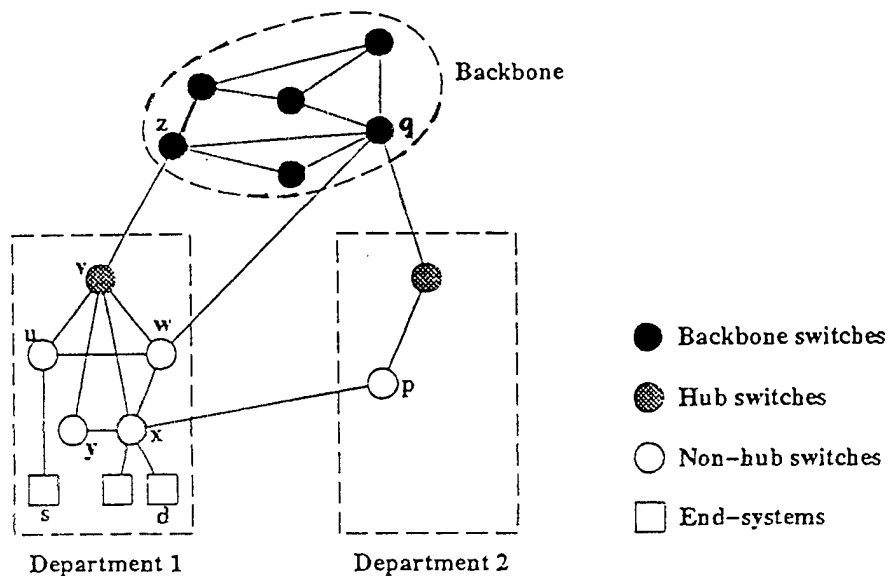
Figure 7: An example network topology.

node u.

Each link has a total of C units of bandwidth.

QoS and Workload Parameters

In our evaluation model, we assume that a VC requires the reservation of a certain amount of bandwidth that is enough to ensure an acceptable QoS for the application. This reservation amount can be thought of either as the peak transmission rate of the VC or its "effective bandwidth" [12] varying between the peak and average transmission rate.

VC setup requests arrive to the network according to a Poisson process of rate λ, each requiring one unit of bandwidth. Each VC, once it is successfully setup, has a lifetime of exponential duration with mean $1/\mu$. The source and the destination end-systems of a VC are chosen randomly.

An arriving VC is admitted to the network if at least one feasible path between its source and destination end-systems is found by the routing protocol, where a feasible path is one that has links with non-zero available capacity. From the set of feasible paths, a minimum hop path is used to establish the VC; one unit of bandwidth is allocated on each of its links for the lifetime of the VC. On the other hand, if a feasible path is not found, then the arriving VC is blocked and lost.

We assume that the available link capacities in the views of the viewservers are updated instan-

taneously whenever a VC is admitted to the network or terminates.

Viewserver Hierarchy Schemes

We have evaluated our viewserver protocol for several different viewserver hierarchies and query methods. We next describe the different viewserver schemes evaluated. Please refer to Figure 7 in the following discussion.

The first viewserver scheme is referred to as base. Each switch is a viewserver. A viewserver's precinct consist of itself and the neighboring nodes. The links in the viewserver's view consist of the links between the nodes in the precinct, and links outgoing from nodes in the precinct to nodes not in the precinct. For example, the precinct of viewserver u consists of nodes u, v, w, s.

As for the viewserver hierarchy, a backbone switch is a level 0 viewserver, a hub switch is a level 1 viewserver and a non-hub switch is a level 2 viewserver. Parent of a hub switch viewserver is the backbone switch viewserver it is connected to. Parent of a non-hub switch viewserver is the hub switch viewserver in its department. Parent of an end-system is the non-hub switch viewserver it is connected to.

We use only one address for each end-system. The viewserver-address of an end-system is the concatenation of four ids. Thus, the address of s is $z.v.u.s$. Similarly, the address of d is $z.v.x.d$. To obtain a route between s and d, it suffices to obtain views of viewservers u, v, x.

The second viewserver scheme is referred to as base-QT (where the QT stands for "query up to top"). It is identical to base except that during the query protocol all the viewservers in the source and the destination addresses are queried. That is, to obtain a route between s and d, the views of u, v, x, z are obtained.

The third viewserver scheme is referred to as vertex-extension. It is identical to base except that viewserver precincts are extended as follows: Let P denote the precinct of a viewserver in the base scheme. For each node u in P, if there is a link from node u to node v and v is not in P, node v is added to the precinct; among v's links, only the ones to nodes in P are added to the view. In the example, nodes z, y, x, q are added to the precinct of u, but outgoing links of these nodes to other nodes are not included (e.g. (x, p) and (z, q) are not included). The advantage of this scheme is that even though it increases the precinct size by a factor of d (where d is the average number of neighbors to a node), it increases the number of links stored in the view by a factor less than 2.

The fourth viewserver scheme is referred to as vertex-extension-QT. It is identical to vertex-extension except that during the query protocol all the viewservers in the source and the destination

addresses are queried.

6 Numerical Results

6.1 Results for Network 1

The parameters of the first network topology, referred to as Network 1, are given in Section 5. The link capacity C is taken to be 20 [6], i.e. a link is capable of carrying 20 VCs simultaneously.

Our evaluation measures were computed for a (randomly chosen but fixed) set of 100,000 VC setup requests. Table 1 lists for each viewserver scheme (1) the minimum, average and maximum of the precinct sizes (in number of nodes), (2) the minimum, average and maximum of the merged view sizes (in number of nodes), and (3) the minimum, average and maximum of the number of viewservers queried.

Scheme	Precinct Size	Merged View Size	No. of Viewservers Queried
base	5 / 16.32 / 28	4 / 56.46 / 81	1 / 5.49 / 6
base-QT	5 / 16.32 / 28	27 / 59.96 / 81	6 / 6.00 / 6
vertex-extension	22 / 88.11 / 288	14 / 155.86 / 199	1 / 5.49 / 6
vertex-extension-QT	22 / 88.11 / 288	113 / 163.28 / 199	6 / 6.00 / 6

Table 1: Precinct sizes, merged view sizes, and number of viewservers queried for Network 1.

The precinct size indicates the memory requirement at a viewserver. More precisely, the memory requirement at a viewserver is $O(\text{precinct size} \times d)$, except for the *vertex-extension* and *vertex-extension-QT* schemes. In these schemes, the memory requirement is increased by a factor less than two. Hence these schemes have the same order of viewserver memory requirement as the *base* and *base-QT* schemes.

The merged view size indicates the memory requirement at a source end-system during the query protocol; i.e. the memory requirement at a source end-system is $O(\text{merged view size} \times d)$ except for the *vertex-extension* and *vertex-extension-QT* schemes. Note that the source end-system does not need to store information about end-systems other than itself and the destination. The numbers in Table 1 take advantage of this.

The number of viewservers queried indicates the communication time required to obtain the merged view at the source end-system. Hence, the "real-time" communication time required to obtain the merged view at a source is slightly more than one round-trip time between the source

168

and the destination.

As is apparent from Table 1, using a QT scheme increases the merged view size by about 6%, and the number of viewservers queried by about 9%. Using the *vertex-extension* scheme increases the merged view size by about 3 times (note that the amount of actual memory needed increases only by a factor less than 2).

The above measures show the memory and time requirements of our protocols. They clearly indicate the savings in storage over the simple approach as manifested by the smaller view sizes. To answer whether the viewserver hierarchy finds many feasible paths, other evaluation measures such as the carried VC load and the percent VC blocking are of interest. They are defined as follows:

- *Carried VC load* is the average number of VCs carried by the network.
- *Percent VC blocking* is the percentage of VC setup requests that are blocked due to the fact that a feasible path is not found.[7]

. In our experiments, we keep the average VC lifetime $(1/\mu)$ fixed at 15000 and vary the arrival rate of VC setup requests (λ). Figure 8 shows the carried VC load versus λ for the simple approach and the viewserver schemes. Figure 9 shows the percent VC blocking versus λ. At low values of λ, all the viewserver schemes are very close to the simple approach. At moderate values of λ, the *base* and *base-QT* schemes perform badly. The *vertex-extension* and *vertex-extension-QT* schemes are still very close to the simple approach (only 3.4% less carried VC load). Note that the performance of the viewserver schemes can be further improved by trying more viewserver addresses.

Surprisingly, at high values of λ, all the viewserver schemes perform better than the simple approach. At $\lambda = 0.5$, the network with the *base* scheme carries about 30% higher load than the simple approach. This is an interesting result. Our explanation is as follows. Elsewhere [2], we have found that when the viewserver schemes can not find an existing feasible path, this path is usually very long (more than 11 hops). This causes our viewserver hierarchy protocols to reject VCs that are admitted by the simple approach over long paths. The use of long paths for VCs is undesirable since it ties up resources at more intermediate nodes, which can be used to admit many shorter length VCs.

In conclusion, we recommend the *vertex-extension* scheme as it performs close to or better than all other schemes in terms of VC carried load and blocking probability over a wide range of workload. Note that for all viewserver schemes, adding QT yields slightly further improvement.

---

[7] Recall that we assume a blocked VC setup request is cleared (i.e. lost).

Figure 8: Carried VC load versus arrival rate for Network 1.
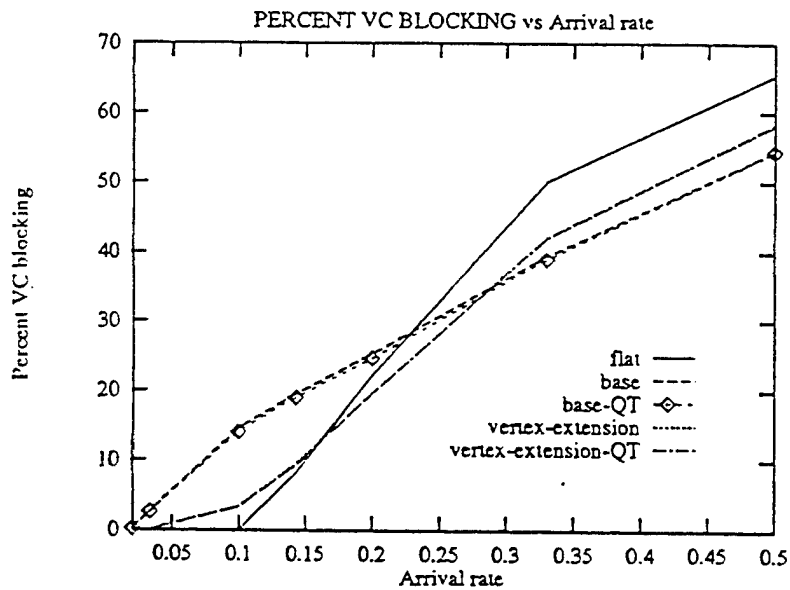
Figure 9: Percent VC blocking versus arrival rate for Network 1.

6.2 Results for Network 2

The parameters of the second network, referred to as Network 2, are the same as the parameters of Network 1. However, a different seed is used for the random number generation, resulting in a different topology and distribution of source-destination end-system pairs for the VCs.

We again take $C = 20$, and we fix $1/\mu$ at 15000. Our evaluation measures were computed for

a set of 100,000 VC setup requests. Table 2, and Figures 10 and 11 show the results. Similar conclusions to Network 1 hold for Network 2. An interesting exception is that at high values of λ, we observe that the *vertex-extension* scheme performs slightly better than the *vertex-extension-QT* scheme (about 4.2% higher carried VC load). The reason is the following: Adding QT gives richer merged views, and hence increases the chance of finding a feasible path that is possibly long. As explained in Section 6.1, this results in performance degradation.

Scheme	Precinct Size	Merged View Size	No. of Viewservers Queried
base	4 / 16.32 / 33	4 / 57.61 / 80	1 / 5.52 / 6
base-QT	4 / 16.32 / 33	30 / 60.64 / 80	6 / 6.00 / 6
vertex-extension	17/ 90.36 / 282	16 / 159.70 / 214	1 / 5.52 / 6
vertex-extension-QT	17 /90.36 / 282	113 / 166.97 / 214	6 / 6.00 / 6

Table 2: Precinct sizes, merged view sizes, and number of viewservers queried for Network 2.

We have repeated the above evaluations for other networks and obtained similar conclusions.

7 Conclusions

We presented a hierarchical VC routing protocol for ATM-like networks. Our protocol satisfies QoS constraints, adapts to dynamic topology changes, and scales well to large number of nodes.

Our protocol uses partial views maintained by viewservers. The viewservers are organized hierarchically. To setup a VC, the source end-system queries viewservers to obtain a merged view that contains itself and the destination end-system. This merged view is then used to compute a source route for the VC.

We evaluated several viewserver hierarchy schemes and compared them to the simple approach. Our results on 2764-node networks indicate that the *vertex-extension* scheme performs close to or better than the simple approach in terms of VC carried load and blocking probability over a wide range of real-time workload. It also reduces the amount of memory requirement by up to two order of magnitude. We note that our protocol scales even better on larger size networks [3].

In all the viewserver schemes we studied, each switch is a viewserver. In practice, not all switches need to be viewservers. We may associate one viewserver with a group of switches. This is particularly attractive in ATM networks where each signaling entity is responsible for establishing VCs across a group of nodes. In such an environment, viewservers and signaling entities can be
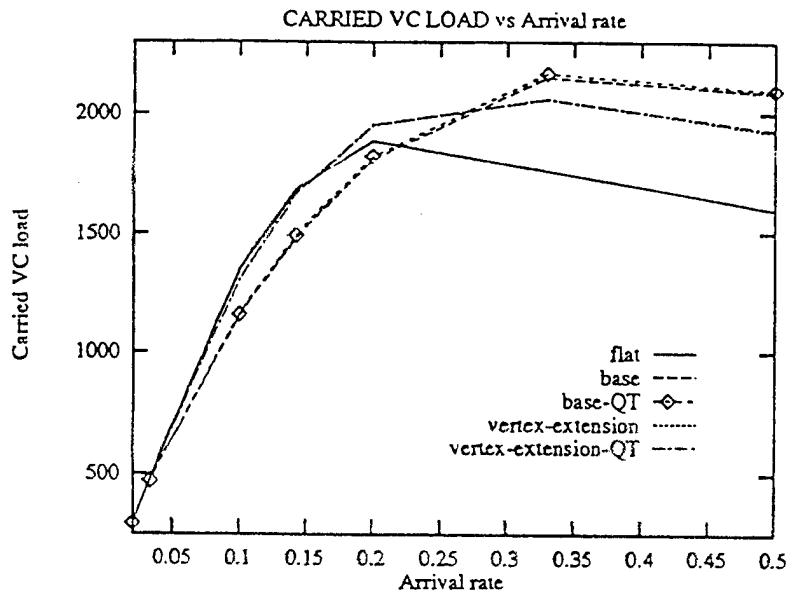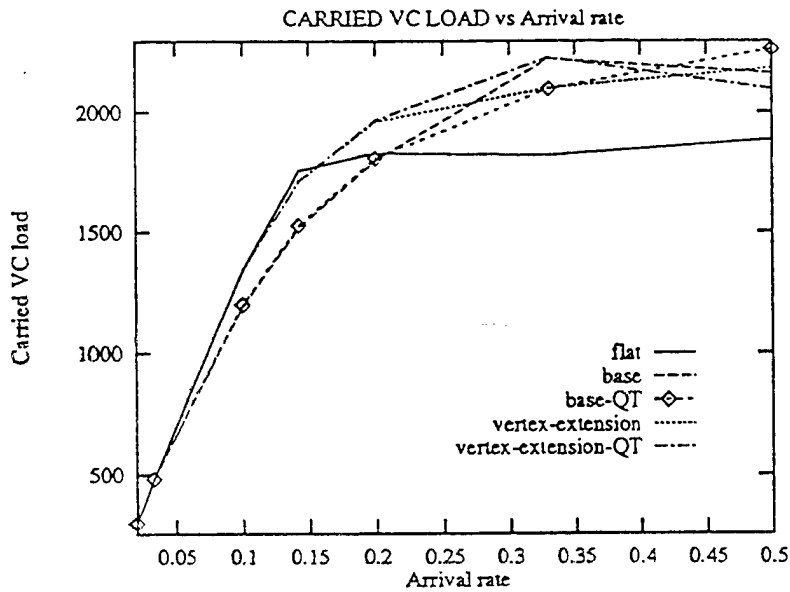
Figure 10: Carried VC load versus arrival rate for Network 2.
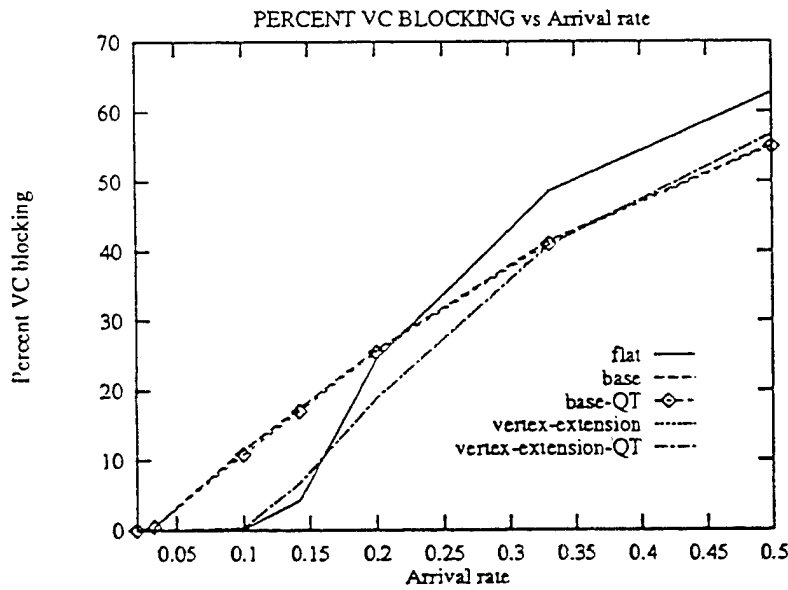
Figure 11: Percent VC blocking versus arrival rate for Network 2.

combined.

However, there is an advantage of each switch being a viewserver; that is, source nodes do not require fixed source routes to their parent viewservers (in the view-query protocol). This reduces the amount of hand configuration required. In fact, the *base* and *base-QT* viewserver schemes do not require any hand configuration.

Our evaluation model assumed that views are instantaneously updated, i.e. no delayed feedback

between link cost changes and view/route changes. We plan to investigate the effect of delayed feedback on the performance of the different schemes. We expect our viewserver schemes to outperform the simple approach in this realistic setting as the update of views of the viewservers requires less time and communication overhead. Thus, views in our viewserver schemes will be more up-to-date.

As we pointed out in [3], the only drawback of our protocol is that to obtain a source route for a VC, views are merged at (or prior to) the VC setup, thereby increasing the setup time. This drawback is not unique to our scheme [8, 16, 7, 11]. Reference [3] describes several ways, including cacheing and replication, to reduce the setup overhead and improve performance.

References

[1] H. Ahmadi, J. Chen, and R. Guerin. Dynamic Routing and Call Control in High-Speed Integrated Networks. In Proc. *Workshop on Systems Engineering and Traffic Engineering, ITC'13*, pages 19–26, Copenhagen, Denmark, June 1991.

[2] C. Alaettinoğlu and A. U. Shankar. Viewserver Hierarchy: A New Inter-Domain Routing Protocol and its Evaluation. Technical Report UMIACS-TR-93-98, CS-TR-3151, Department of Computer Science, University of Maryland, College Park, October 1993. Earlier version CS-TR-3033, February 1993.

[3] C. Alaettinoğlu and A. U. Shankar. Viewserver Hierarchy: A New Inter-Domain Routing Protocol. In Proc. *IEEE INFOCOM '94*, Toronto, Canada, June 1994.

[4] A. Alles. ATM in Private Networking: A Tutorial. Hughes LAN Systems, 1993.

[5] P. Almquist. Type of Service in the Internet Protocol Suite. Technical Report RFC-1349, Network Working Group, July 1992.

[6] L. Breslau, D. Estrin, and L. Zhang. A Simulation Study of Adaptive Source Routing in Integrated Services Networks. Available by anonymous ftp at catarina.usc.edu:pub/breslau, September 1993.

[7] J. N. Chiappa. A New IP Routing and Addressing Architecture. Big-Internet mailing list., 1992. Available by anonymous ftp from munnari.oz.au:big-internet/list-archive.

[8] D.D. Clark. Policy routing in Internet protocols. Request for Comment RFC-1102, Network Information Center, May 1989.

[9] R. Coltun and M. Sosa. VC Routing Criteria. Internet Draft, March 1993.

[10] D. Comer and R. Yavatkar. FLOWS: Performance Guarantees in Best Effort Delivery Systems. In Proc. *IEEE INFOCOM, Ottawa, Canada*, pages 100–109, April 1989.

[11] D. Estrin, Y. Rekhter, and S. Hotz. Scalable Inter-Domain Routing Architecture. In Proc. *ACM SIGCOMM '92*, pages 40–52, Baltimore, Maryland, August 1992.

[12] R. Guerin, H. Ahmadi, and M. Naghshineh. Equivalent Capacity and its Application to Bandwidth Allocation in High-Speed Networks. *IEEE J. Select. Areas Commun.*, SAC-9(7):968–981, September 1991.

[13] A. Guillen, R. Kia, and B. Sales. An Architecture for Virtual Circuit/QoS Routing. In Proc. *IEEE International Conference on Network Protocols '93*, pages 80–87, San Francisco, California, October 1993.

[14] S. Gupta, K. Ross, and M. ElZarki. Routing in Virtual Path Based ATM Networks. In Proc. *GLOBECOM '92*, pages 571–575, 1992.

[15] R-H. Hwang, J. Kurose, and D. Towsley. MDP Routing in ATM Networks Using Virtual Path Concept. In Proc. *IEEE INFOCOM*, pages 1509–1517, Toronto, Ontario, Canada, June 1994.

[16] M. Lepp and M. Steenstrup. An Architecture for Inter-Domain Policy Routing. Internet Draft. Available from the authors., June 1992.

[17] I. Matta and A.U. Shankar. An Iterative Approach to Comprehensive Performance Evaluation of Integrated Services Networks. In Proc. *IEEE International Conference on Network Protocols '94*, Boston, Massachusetts, October 1994. To appear.

[18] J. Moy. OSPF Version 2. RFC 1247, Network Information Center, SRI International, July 1991.

[19] C. Parris and D. Ferrari. A Dynamic Connection Management Scheme for Guaranteed Performance Services in Packet-Switching Integrated Services Networks. Technical Report TR-93-005, International Computer Science Institute, Berkeley, California, January 1993.

[20] J. Postel. Transmission Control Protocol: DARPA Internet Program Protocol Specification. Request for Comment RFC-793, Network Information Center, SRI International, 1981.

[21] M. Prycker. *Asynchronous Transfer Mode - Solution for Broadband ISDN*. Ellis Horwood, 1991.

[22] S. Rampal, D. Reeves, and D. Agrawal. An Evaluation of Routing and Admission Control Algorithms for Multimedia Traffic in Packet-Switched Networks. Available from the authors, 1994.

[23] H. Suzuki and F. Tobagi. Fast Bandwidth Reservation Scheme with Multi-Link and Multi-Path Routing in ATM Networks. In Proc. *IEEE INFOCOM '92*, pages 2233–2240, Florence, Italy, May 1992.

[24] E. Sykas, K. Vlakos, I. Venieris, and E. Protonotarios. Simulative Analysis of Optimal Resource Allocation and Routing in IBCN's. *IEEE J. Select. Areas Commun.*, 9(3):486–492, April 1991.

[25] P. F. Tsuchiya. The Landmark Hierarchy: Description and Analysis, The Landmark Routing: Architecture Algorithms and Issues. Technical Report MTR-87W00152, MTR-87W00174, The MITRE Corporation, McLean, Virginia, 1987.

[26] P. F. Tsuchiya. The Landmark Hierarchy:A New Hierarchy For Routing In Very Large Networks. In Proc. *ACM SIGCOMM '88*, August 1988.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE October 1994	3. REPORT TYPE AND DATES COVERED Technical

4. TITLE AND SUBTITLE

A Scalable Virtual Circuit Routing Scheme for ATM Networks

5. FUNDING NUMBERS

C DASG60-92-0055
G NCR 89-04590
G NCR 93-21043

6. AUTHOR(S)

Cengiz Alaettinoglu, Ibrahim Matta and A. Udaya Shankar

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Department of Computer Science
A. V. Williams Building
University of Maryland
College Park, MD 20742

8. PERFORMING ORGANIZATION REPORT NUMBER

CSTR-3360
UMIACS-TR 94-115

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Phillips Laboratory
Directorate of Contracting
3651 Lowry Avenue SE
Kirtland AFB NM 87117-5777

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

High-speed networks, such as ATM networks, are expected to support diverse quality-of-service (QoS) requirements, including real-time QoS. Real-time QoS is required by many applications such as voice and video. To support such service, routing protocols based on the Virtual Circuit (VC) model have been proposed. However, these protocols do not scale well to large networks in terms of storage and communication overhead.

In this paper, we present a scalable VC routing protocol. It is based on the recently proposed viewserver hierarchy, where each viewserver maintains a partial view of the network. By querying these viewservers, a source can obtain a merged view that contains a path to the destination. The source then sends a request packet over this path to setup a real-time VC through resource reservations. The request is blocked if the setup fails. We compare our protocol to a simple approach using simulation. Under this simple approach, a source maintains a full view of the network. In addition to the savings in storage, our results indicate that our protocol performs close or better than the simple approach in terms of VC carried load and blocking probability over a wide range of real-time workload.

14. SUBJECT TERMS

Computer-Communication Networks: Network Architecture and Design, Network Protocols; Routing Protocols: Computer Network Routing Protocols

15. NUMBER OF PAGES
22 pages

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT Unlimited

Hierarchical Inter-Domain Routing Protocol
with On-Demand ToS and Policy Resolution*

Cengiz Alaettinoğlu, A. Udaya Shankar

Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
College Park, Maryland 20742

June 20, 1994

Abstract

Traditional inter-domain routing protocols based on superdomains maintain either "strong" or "weak" ToS and policy constraints for each visible superdomain. With strong constraints, a valid path may not be found even though one exists. With weak constraints, an invalid domain-level path may be treated as a valid path.

We present an inter-domain routing protocol based on superdomains, which always finds a valid path if one exists. Both strong and weak constraints are maintained for each visible superdomain. If the strong constraints of the superdomains on a path are satisfied, then the path is valid. If only the weak constraints are satisfied for some superdomains on the path, the source uses a query protocol to obtain a more detailed "internal" view of these superdomains, and searches again for a valid path. Our protocol handles topology changes, including node/link failures that partition superdomains. Evaluation results indicate our protocol scales well to large internetworks.

Categories and Subject Descriptors: C.2.1 [Computer-Communication Networks]: Network Architecture and Design—*packet networks; store and forward networks*; C.2.2 [Computer-Communication Networks]: Network Protocols—*protocol architecture*; C.2.m [Routing Protocols]; F.2.m [Computer Network Routing Protocols].

---

type-of-service (ToS) constraints of applications (e.g. low delay, high throughput, high reliability, minimum monetary cost), each node maintains a *cost* for each outgoing link and ToS. The intra-domain routing protocol should choose optimal paths based on these costs.

Across all domains, an *inter-domain routing protocol* is executed that provides routes between source and destination nodes in different domains, using the services of the intra-domain routing protocols within domains. This protocol should have the following properties:

(1) It should satisfy the policy constraints of domains. To do this, it must keep track of the policy constraints of domains [5].

(2) An inter-domain routing protocol should also satisfy ToS constraints of applications. To do this, it must keep track of the ToS services offered by domains [5].

(3) An inter-domain routing protocol should scale up to very large internetworks, i.e. with a very large number of domains. Practically this means that processing, memory and communication requirements should be *much less than linear* in the number of domains. It should also handle non-hierarchical domain interconnections at any level [8] (e.g. we do not want to hand-configure special routes as "back-doors").

(4) An inter-domain routing protocol should automatically adapt to link cost changes and node/link failures and repairs, including failures that partition domains [13].

A Straight-Forward Approach

A straight-forward approach to inter-domain routing is domain-level source routing with link-state approach [7, 5]. In this approach, each router[3] maintains a *domain-level view* of the internetwork, i.e., a graph with a vertex for every domain and an edge between every two neighbor domains. Policy and ToS information is attached to the vertices and the edges of the view.

When a source node needs to reach a destination node, it (or a router[4] in the source's domain) first examines this view and determines a *domain-level source route* satisfying ToS and policy constraints, i.e., a sequence of domain ids starting from the source's domain and ending with the destination's domain. Then packets are routed to the destination using this domain-level source route and the intra-domain routing protocols of the domains crossed.

For example, consider the internetwork of Figure 2 (each circle is a domain, and each thin line

---

[3] Not all nodes maintain routing tables. A router is a node that maintains a routing table.

[4] referred to as the policy server in [7]

1 Introduction

A computer internetwork, such as the Internet, is an interconnection of backbone networks, regional networks, metropolitan area networks, and stub networks (campus networks, office networks and other small networks)[1]. Stub networks are the producers and consumers of the internetwork traffic, while backbones, regionals and MANs are transit networks. Most of the networks in an internetwork are stub networks. Each network consists of nodes (hosts, routers) and links. A node that has a link to a node in another network is called a *gateway*. Two networks are *neighbors* when there is one or more links between gateways in the two networks (see Figure 1).
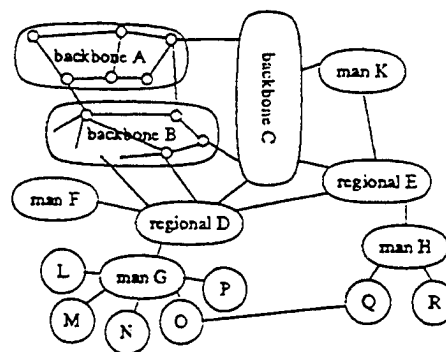
Figure 1: A portion of an internetwork. (Circles represent stub networks.)

An internetwork is organized into *domains*[2]. A domain is a set of networks (possibly consisting of only one network) administered by the same agency. Domains are typically subject to *policy constraints*, which are administrative restrictions on inter-domain traffic [7, 11, 8, 5]. The policy constraints of a domain U are of two types: *transit policies*, which specify how other domains can use the resources of U (e.g. $0.01 per packet, no traffic from domain V); and *source policies*, which specify constraints on traffic originating from U (e.g. domains to avoid/prefer, acceptable connection cost). Transit policies of a domain are public (i.e. available to other domains), whereas source policies are usually private.

Within each domain, an *intra-domain routing protocol* is executed that provides routes between source and destination nodes in the domain. This protocol can be any of the typical ones, i.e., next-hop or source routes computed using distance-vector or link-state algorithms. To satisfy

---

[1] For example, NSFNET, MILNET are backbones, and Suranet, CerfNet are regionals.
[2] Also referred to as *routing domains* or *administrative domains*.

is a domain-level interconnection). Suppose a node in $d1$ desires a connection to a node in $d7$. Suppose the policy constraints of $d3$ and $d19$ do not allow transit traffic originating from $d1$. Every node maintains this information in its view. Thus the source node can choose a valid path from source domain $d1$ to destination domain $d7$ avoiding $d3$ and $d19$ (e.g. thick line in the figure).
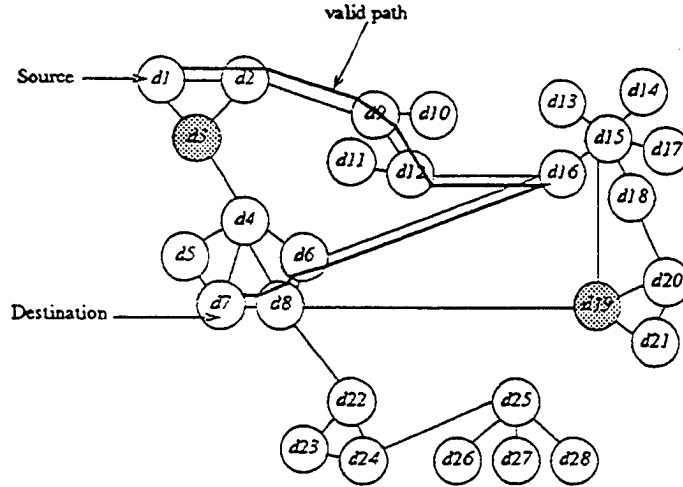
Figure 2: An example interdomain topology.

The disadvantage of this straightforward scheme is that it does not scale up for large internetworks. The storage at each router is proportional to $N_D \times E_D$, where N_D is the number of domains and E_D is the average number of neighbor domains to a domain. The communication cost for updating views is proportional to $N_R \times E_R$, where N_R is the number of routers in the internetwork and E_R is the average router neighbors of a router (topology changes are flooded to all routers in the internetwork).

The Superdomain Approach

To achieve scaling, several approaches based on hierarchically aggregating domains into *superdomains* have been proposed [16, 14, 6]. Here, each domain is a level 1 superdomain, "close" level 1 superdomains are grouped into level 2 superdomains, "close" level 2 superdomains are grouped into level 3 superdomains, and so on (see Figure 3). Each router x maintains a view that contains the level 1 superdomains in x's level 2 superdomain, the level 2 superdomains in x's level 3 superdomain (excluding the x's level 2 superdomain), and so on. Thus a router maintains a smaller view than it would in the absence of hierarchy. For the superdomain hierarchy of Figure 3, the views of two

Figure 3: An example of superdomain hierarchy.

routers (one in domain $d1$ and one in domain $d16$) are shown in Figures 4 and 5.

Figure 4: View of a router in $d1$.

Figure 5: View of a router in $d16$.

The superdomain approach has several problems. One problem is that the aggregation results in loss of domain-level ToS and policy information. A superdomain is usually characterized by a single set of ToS and policy constraints derived from the ToS and policy constraints of the domains in it. Routers outside the superdomain assume that this set of constraints applies uniformly to each of its children (and by recursion to each domain in the superdomain). If there are domains with different (possibly contradictory) constraints in a superdomain, then there is no good way of deriving the ToS and policy constraints of the superdomain.

The usual technique [16] of obtaining ToS and policy constraints of a superdomain is to obtain either a *strong* set of constraints or a *weak* set of constraints[5] from the ToS and policy constraints of

---
[5] "strong" and "weak" are referred to respectively as "union" and "intersection" in [16]

181

the children superdomains in it. If strong (weak) constraints are used for policies, the superdomain enforces a policy constraint if that policy constraint is enforced by some (all) of its children. If strong (weak) constraints are used for ToS constraints, the superdomain is assumed to support a ToS if that ToS is supported by all (some) of its children. The intention is that if strong (weak) constraints of a superdomain are (are not) satisfied then any (no) path through that superdomain is valid.

Each approach has problems. Strong constraints can eliminate valid paths, and weak constraints can allow invalid paths. For example in Figure 3, $d16$ allows transit traffic from $d1$ while $d19$ does not; with strong constraints G would not allow transit traffic from $d1$, and with weak constraints G would allow transit traffic from $d1$ to be routed via $d19$.

Other problems of the superdomain approach are that the varying visibilities of routers complicates superdomain-level source routing and handling of node/link failures (especially those that partition superdomains). The usual technique for solving these problems is to augment superdomain-level views with gateways [16] (see Section 3).

Our Contribution

In this paper, we present an inter-domain routing protocol based on superdomains, which finds a valid path if and only if one exists. Both strong and weak constraints are maintained for each visible superdomain. If the strong constraints of the superdomains on a path are satisfied, then the path is valid. If only the weak constraints are satisfied for some superdomains on the path, the source uses a query protocol to obtain a more detailed "internal" view of these superdomains, and searches again for a valid path.

We use superdomain-level views with gateways and a link-state view update protocol to handle topology changes including failures that partition superdomains. The storage cost is $O(\log N_D \times \log N_D)$ without the query protocol. We demonstrate the scaling properties of the query protocol by giving evaluation results based on simulations. Our evaluation results indicate that the query protocol can be performed using 15% extra space.

Our protocol consists of two subprotocols: a **view-query protocol** for obtaining views of greater resolution when needed; and a **view-update protocol** for disseminating topology changes to the views.

182

Several approaches to scalable inter-domain routing have been proposed, based on the super-domain hierarchy [1, 14, 16, 9, 6], and the landmark hierarchy [18, 17]. Some of these approaches suffer from loss of ToS and policy information (and hence may not find a valid path which exists). Others are still in a preliminary stage. (Details in Section 8.)

One important difference between these approaches and ours is that ours uses a query mechanism to obtain ToS and policy details whenever needed. In our opinion, such a mechanism is needed to obtain a scalable solution. Query protocols are also being developed to enhance the protocols in [9, 6]. Reference [2] presents protocols based on a new kind of hierarchy, referred to as the viewserver hierarchy (more details in Section 8).

A preliminary version of the view-query protocol was proposed in reference [1]. That version differs greatly from the one in this paper. Here, we augment superdomain-level views with gateways. In [1], we augmented superdomain-level views with superdomain-to-domain edges (details in Section 8). Both versions have the same time and space complexity, but the protocols in this paper are much simpler conceptually. Also the view-update protocol is not in reference [1].

Organization of the paper

In Section 2, we present some definitions used in this paper. In Section 3, we define the view data structures. In Section 4, we describe how views are affected by topology changes. In Section 5, we present the view-query protocol. In Section 6, we present the view-update protocol. In Section 7, we present our evaluation model and the results of its application to the superdomain hierarchy. In Section 8, we survey recent approaches to inter-domain routing. In Section 9, we conclude and describe cacheing and heuristic schemes to improve performance.

2 Preliminaries

Each domain has a unique id. Let DomainIds denote the set of domain-ids. Each node has a unique id. Let NodeIds denote the set of node-ids. For a node x, we use domainid(x) to denote the domain-id of x's domain.

The superdomain hierarchy defines the following parent-child relationship: a level i, $i > 1$, superdomain is the parent of each level $i - 1$ superdomain it contains. Top-level superdomains

have no parents. Level 1 superdomains, which are just domains, have no children. For any two superdomains X and Y, X is a sibling of Y iff X and Y have the same parent. X is an ancestor (descendant) of Y iff $X = Y$ or X is an ancestor (descendant) of Y's parent (child).

Each router maintains information about a subset of superdomains, referred to as its visible superdomains. The *visible superdomains* of a router x are (1) x's domain itself, (2) siblings of x's domain, and (3) siblings of ancestors of x's domain. In Figure 3, the visible superdomains of a router in $d1$ are $d1, d2, d3, B, C, G, J$ (these are shown in Figure 4). Note that if a superdomain U is visible to a router, then no ancestor or descendant of U is visible to the router.

Each superdomain has a unique id, i.e. unique among all superdomains regardless of level. Let SuperDomainIds denote the set of superdomain-ids. DomainIds is a subset of SuperDomainIds. For a superdomain U, let level(U) denote the level of U in the hierarchy, let Ancestors(U) denote the set of ids of ancestor superdomains of U in the hierarchy, and let Children(U) denote the set of ids of child superdomains of U in the hierarchy.

For a router x, let VisibleSuperDomains(x) denote the set of ids of superdomains visible from x.

We extend the above definitions by allowing their arguments to be nodes, in which case the node stands for its domain. For example, if x is a node in domain d, Ancestors(x) denotes Ancestors(d).

3 Superdomain-Level Views with Gateways

For routing purposes, each domain (and node) has an address, defined as the concatenation of the superdomain ids starting from the top level and going down to the domain (node). For example in Figure 3, the address of domain $d15$ is $G.E.d15$, and the address of a node h in $d15$ is $G.E.d15.h$.

When a source node needs to reach a destination node, it first determines the visible superdomain in the destination address and then by examining its view determines a superdomain-level source route (satisfying ToS and policy constraints) to this superdomain. However, since routers in different superdomains maintain views of different sets of superdomains, this superdomain-level source route can be meaningless at some intermediate superdomain's router x because the next superdomain in this source route is not visible to x. For example in Figure 4, superdomain-level source route $\langle d2, B, G, C \rangle$ created at a router in $d2$ becomes meaningless once the packet is in G, where C is not visible.

The usual technique of solving this problem is to augment superdomain-level views with gateways and edges between these gateways.

Define the pair $U{:}g$ to be an *sd-gateway* iff U is a superdomain and g is a node that is in U and has a link to a node outside U. Equivalently, we say that g is *a gateway of* U.

Define $\langle U{:}g, h \rangle$ to be an *actual-edge* iff $U{:}g$ is an sd-gateway, h is a gateway not in U, and there is a link from g to h.

Define $\langle U{:}g, h \rangle$ to be a *virtual-edge* iff $U{:}g$ and $U{:}h$ are sd-gateways and $g \neq h$ (note that there may not be a link between g and h).

$\langle U{:}g, h \rangle$ is an *edge* iff it is an actual-edge or a virtual-edge. An edge $\langle U{:}g, h \rangle$ is also said to be an *outgoing edge* of $U{:}g$. Define *edges of* $U{:}g$ to be the set of edges outgoing from $U{:}g$. Define *edges of* U to be the set of edges outgoing from any gateway of U.

Let $\texttt{Gateways}(U)$ denote the set of node-ids of gateways of U. Let $\texttt{Edges}(U{:}g)$ denote the edges of $U{:}g$. Note that we never use "edge" as a synonym for link.

A gateway g of a domain can generate many sd-gateways, specifically, $U{:}g$ for every ancestor U of g's domain such that g has a link to a node outside U. A link $\langle g, h \rangle$ where g and h are gateways in different domains, can generate many actual-edges; specifically, actual-edge $\langle U{:}g, h \rangle$ for every ancestor U of g's domain such that U is not an ancestor of h's domain.

For the internetwork topology of Figure 2, the corresponding gateway-level connections are shown in Figure 6 where black rectangles are gateways. For the hierarchy of Figure 3, gateway g in Figure 6 generates sd-gateways $d16{:}g$, $E{:}g$, and $G{:}g$. The link $\langle g, h \rangle$ in Figure 6 generates actual-edges $\langle d16{:}g, h \rangle$, $\langle E{:}g, h \rangle$, $\langle G{:}g, h \rangle$.

To a router, at most one of the sd-gateways generated by a gateway g is visible, namely $U{:}g$ where U is an ancestor of g's domain and U is visible to the router. At most one of the actual-edges generated by a link $\langle g, h \rangle$ between two gateways in different domains is visible to the router, namely edge $\langle U{:}g, h \rangle$ where $U{:}g$ is visible to the router. None of the actual-edges are visible to the router if g and h are inside a visible superdomain. For example in Figure 3, of the actual-edges generated by link $\langle g, h \rangle$, only $\langle G{:}g, h \rangle$ is visible to a router in $d1$, and only $\langle d16{:}g, h \rangle$ is visible to a router in $d16$.

A router maintains a view consisting of the visible sd-gateways and their outgoing actual- and virtual-edges. An edge $\langle U{:}g, h \rangle$ in the view of a router connects the sd-gateway $U{:}g$ to the sd-
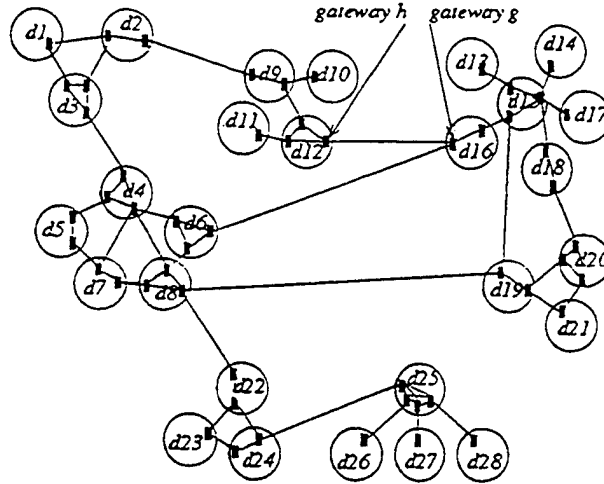
Figure 6: Gateway-level connections of internetwork of Figure 2.

gateway $V:h$ such that $V:h$ is visible to the router. For the superdomain-level views of Figures 4 and 5, the new views are shown in Figures 7 and 8, respectively.
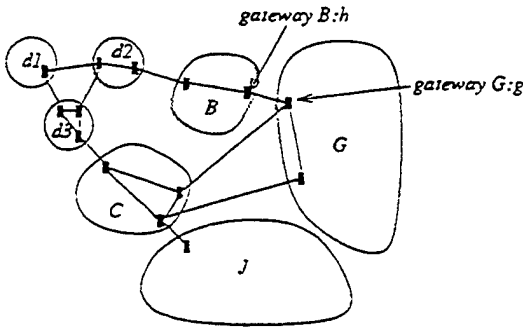
Figure 7: View of a router in $d1$.
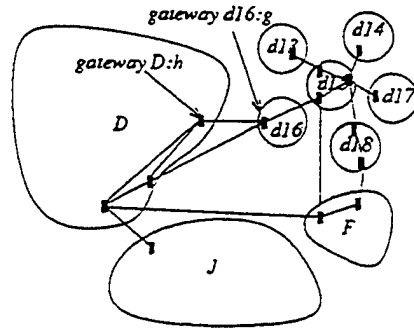
Figure 8: View of a router in $d16$.

The view of a router x contains, for each superdomain U that is visible to x or is an ancestor of x, the strong and weak constraints of U and a set referred to as $Gateways\&Edges_x(U)$. This set contains, for each gateway y of U, the edges of $U:y$ and their costs. The reason for storing information about ancestor superdomains is given in Section 5. The cost field is used to satisfy ToS constraints and is described in Section 4. The $timestamp$ field is described in Section 6. Formally, the view of x is defined as follows:

186

$View_x$. View of x.

$$= \{\langle U, \texttt{strong\_constraints}(U), \texttt{weak\_constraints}(U), \mathit{Gateways\&Edges_x}(U)\rangle :$$
$$U \in \texttt{VisibleSuperDomains}(x) \cup \texttt{Ancestors}(x) \}$$

where

$\mathit{Gateways\&Edges_x}(U)$. Sd-gateways and edges of U.
$$= \{\langle y, \mathit{timestamp}, \{\langle z, \mathit{cost}\rangle : \langle U{:}y, z\rangle \in \texttt{Edges}(U{:}y)\}\rangle : y \in \texttt{Gateways}(U) \}.$$

ToS and policy constraints can also be specified for each sd-gateway and edge. Our protocols can be extended to handle such constraints, but we have not done so here in order to keep their descriptions simple.

A *superdomain-level source route* is now a sequence of sd-gateway ids. With this definition, it is easy to verify that whenever the next superdomain in a superdomain-level source route is not visible to a router, there is an actual-edge (hence a link) between the router and the next gateway in this route.

4   Edge-Costs and Topology Changes

A cost is associated with each edge. The cost of an edge equals a vector of values if the edge is up; each cost value indicates how expensive it is to cross the edge according to some ToS constraint. The cost equals ∞ if the edge is an actual-edge and it is down, or the edge is a virtual-edge $\langle U{:}g, h\rangle$ and h can not be reached from g without leaving U.

Since an actual-edge represents a physical link, its cost can be determined from measured link statistics. The cost of a virtual-edge $\langle U{:}g, h\rangle$ is an aggregation of the cost of physical links in U and is calculated as follows: If U is a domain, the cost of $\langle U{:}g, h\rangle$ is calculated as the maximum/minimum/average cost of the routes within U from g to h [4]. For higher level superdomains U, the cost of $\langle U{:}g, h\rangle$ is derived from the costs of edges between the gateways of children superdomains of U.

Link cost changes and link/node failures and repairs correspond to cost changes, failures and repairs of actual- and virtual-edges. Thus the attributes of edges in the views of routers must be regularly updated. For this, we employ a view-update protocol (see Section 6).

Link/node failures can also partition a superdomain into cells, where a *cell* of a superdomain is defined to be a maximal subset of nodes of the superdomain that can reach each other without leaving the superdomain. Superdomain partitions can occur at any level in the hierarchy. For example, suppose U is a domain and V is its parent superdomain. U can be partitioned into cells without V being partitioned (i.e. if the cells of U can reach each other without leaving V). The opposite can also happen: if all links between U and the other children of V fail, then V becomes partitioned but U does not. Or both U and V can be partitioned. In the same way, link/node repairs can merge cells into bigger cells.

We handle superdomain partitioning as follows: A router detects that a superdomain U is partitioned when a virtual-edge of U in the router's view has cost ∞. When a router forwards a packet to a destination for which the visible superdomain, say U, in the destination address is partitioned into cells, a copy of the packet is sent to each cell by sending a copy of the packet to each gateway of U; the id U in the destination address is "marked" in the packet so that subsequent routers do not create new copies of the packet for U.

5 View-Query Protocol

When a source node wants a superdomain-level source route to a destination, a router in its domain examines its view and searches for a valid path (i.e. superdomain-level source route) using the destination address[6]. We refer to this router as the *source router*. Even though the source router does not know the constraints of the individual domains that are to be crossed in each superdomain, it does know the strong and weak constraints of the superdomains. We refer to a superdomain whose strong constraints are satisfied as a *valid superdomain*. If a superdomain's weak constraints are satisfied but strong constraints are not satisfied, then there may be a valid path through this superdomain. We refer to such a superdomain as a *candidate superdomain*.

A path is valid if it involves only valid superdomains. A path cannot be valid if it involves a superdomain which is neither valid nor candidate. We refer to a path involving only valid and candidate superdomains as a *candidate path*.

---

[6] We assume that the source has the destination's address. If that is not the case, it would first query the name servers to obtain the address for the destination. Querying the name servers can be done the same way it is done currently in the Internet. It requires nodes to have a set of fixed addresses to name servers. This is also sufficient in our case.

188

If the source router's view contains a candidate path $\langle U_0{:}g_{0_0}, \ldots, U_0{:}g_{0_{n_0}}, U_1{:}g_{1_0}, \ldots, U_1{:}g_{1_{n_1}}, \cdots,$ $U_m{:}g_{m_0}, \ldots, U_m{:}g_{m_{n_m}} \rangle$ to the destination (and does not contain a valid path), then for each candidate superdomain U_i on this path, the source router queries gateway g_{i_0} of U_i for the internal view of U_i. This internal view consists of the constraints, sd-gateways and edges of the child superdomains of U_i.

When a router x receives a request for the internal view of an ancestor superdomain U, it returns the following data structure:

$IView_x(U)$. Internal view of U at router x.

$= \{\langle V, \texttt{strong\_constraints}(V), \texttt{weak\_constraints}(V), Gateways\&Edges_x(V)\rangle \in View_x :$
$\qquad V \in \texttt{Children}(U)\}$

It is to simplify the construction of $IView_x(U)$ that we store information about ancestor superdomains in the view of router x. Instead of storing this information, router x could construct $IView_x(U)$ from the constraints, sd-gateways and edges of the visible descendants of U. We did not choose this alternative because the extra information does not increase storage complexity.

When the source router receives the internal view of a superdomain U, it does the following: (1) it removes the sd-gateways and edges of U from its view; (2) it adds the sd-gateways and edges of children superdomains in the internal view of U; and (3) searches for a valid path again. If there is still no valid path but there are candidate paths, the process is repeated.

For example, consider Figure 3. For a router in superdomain $d1$ (see Figure 7), G is visible and is a candidate domain. The internal view of G is shown in Figure 9, and the resulting merged view is shown in Figure 10. The valid path through G (visiting $d16$ and avoiding $d19$) can be discovered using this merged view (since the strong constraints of E are satisfied).

Consider a candidate route to a destination: $\langle U_0{:}g_{0_0}, \ldots, U_0{:}g_{0_{n_0}}, U_1{:}g_{1_0}, \ldots, U_1{:}g_{1_{n_1}}, \cdots,$ $U_m{:}g_{m_0}, \ldots, U_m{:}g_{m_{n_m}} \rangle$. If superdomain U_i is partitioned into cells, it may re-appear later in the candidate path (i.e. for some $j \neq i$, $U_j = U_i$). In this case both gateways g_{i_0} and g_{j_0} are queried. Timestamps are used to resolve conflicts between the information reported by these gateways.

The view-query protocol uses two types of messages as follows:

- $(\texttt{RequestIView}, sdid, gid, s\_address, d\_address)$

189
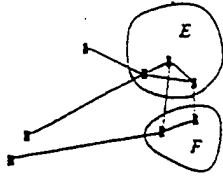
Figure 9: Internal view of G.
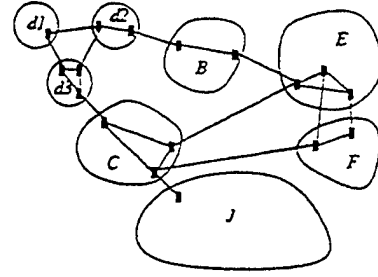
Figure 10: Merged view at $d1$.

Sent by a source router to gateway gid to obtain the internal view of superdomain $sdid$. $s\_address$ is the address of the source router. $d\_address$ is the address of the destination node (of the desired route).

- (ReplyIView, $sdid, gid, iview, d\_address$)

  where $iview$ is the internal view of superdomain $sdid$, and other parameters are as in the RequestIView message. It is sent by gateway gid to the source router.

The state maintained by a source router x is listed in Figure 15. $PendingReq_x$ is used to avoid sending new request messages before receiving all outstanding reply messages. $WView_x$ and $PendingReq_x$ are allocated and deallocated on demand for each destination.

The events of router x are specified in Figure 15. In the figure, * is a wild-card matching any value. $TimeOut_x$ event is executed after a time-out period from the execution of $Request_x$ event to indicate that the request has not been satisfied. The source host can then repeat the same request afterwards.

The procedure $search_x$ uses an operation "ReliableSend(m) to v", where m is the message being sent and v is either an address of an arbitrary router or an id of a gateway of a visible superdomain. ReliableSend is asynchronous. The message is delivered to v as long as there is a sequence of up links between u and v.[7] (Note that an address is not needed to obtain an inter-domain route to a gateway of a visible superdomain.)

Router Failure Model: A router can undergo failures and recoveries at anytime. We assume failures are fail-stop (i.e. a failed router does not send erroneous messages). When a router x recovers, the variables $WView_x$ and $PendingReq_x$ are lost for all destinations. The cost of each edge in $View_x$ is set to ∞. It becomes up-to-date as the router receives new information from other

---

[7] This involves time-outs, retransmissions, etc. It requires a transport protocol support such as TCP.

routers.

6 View-Update Protocol

A gateway g, for each ancestor superdomain U, informs other routers of topology changes (i.e. failures, repairs and cost changes) affecting U:g's edges. The communication is done by flooding messages. The flooding is restricted to the routers in the parent superdomain of U, since U is visible only to these routers.

Due to the nature of flooding, a router can receive information out of order from a gateway. In order to avoid old information replacing new information, each gateway includes increasing time stamps in the messages it sends. Routers maintain for each gateway the highest received time stamp (in the *timestamp* field in $View_x$), and discard messages with smaller timestamps. Time stamps do not have to be real-time clock values.

Due to superdomain partitioning, messages sent by a gateway may not reach all routers within the parent superdomain, resulting in some routers having out-of-date information. This out-of-date information can cause inconsistencies when the partition is repaired. To eliminate inconsistencies, when a link recovers, the two routers at the ends of the link exchange their views and flood any new information. As usual, information about a superdomain U is flooded over U's parent superdomain.

The view-update protocol uses messages of the following form:

- (Update, *sdid*, *gid*, *timestamp*, *edge-set*)

  Sent by the gateway *gid* to inform other routers about current attributes of edges of *sdid*:*gid*. *timestamp* indicates the time stamp of *gid*. *edge-set* contains a cost for each edge.

The state maintained by a router x is listed in Figure 16. Note that AdjLocalRouters$_x$ or AdjForeignGateways$_x$ can be empty. $IntraDomainRT_x$ contains a route (next-hop or source)[8] for every reachable node of the domain. We assume that consecutive reads of $Clock_x$ returns increasing values.

Routers also receive and flood messages containing edges of sd-gateways of their ancestor superdomains. This information is used by the query protocol (see Section 5). Also the highest timestamp received from a gateway g of an ancestor superdomain is needed to avoid exchanging

---

[8] $IntraDomainRT_x$ is a view in case of a link-state routing protocol or a distance table in case of a distance-vector routing protocol.

the messages of g infinitely during flooding.

The events of router x are specified in Figure 16. We use $Ancestor_i(U)$ to denote the superdomain-id of the ith ancestor of U, where $Ancestor_0(U) = U$. In the view-update protocol, a node u uses send operations of the form "Send(m) to v", where m is the message being sent and v is the destination-id. Here, nodes u and v are neighbors, and the message is sent over the physical link $\langle u, v \rangle$. If the link is down, we assume that the packet is dropped.

7 Evaluation

In the superdomain hierarchy (without the query protocol), the number of superdomains in a view is logarithmic in the number of superdomains in the internetwork [10].[9] However, the storage required for a view is proportional not to the number of superdomains in it but to the number of sd-gateways in it. As we have seen, there can be more than one sd-gateway for a superdomain in a view.

In fact, the superdomain hierarchy does not scale-up for arbitrary internetworks; that is, the number of sd-gateways in a view can be proportional to the number of domains in the internetwork. For example, if each domain in a superdomain U has a distinct gateway with a link to outside U, the number of sd-gateways of U would be linear in the number of domains in U.

The good news is that the superdomain hierarchy does scale-up for realistic internetwork topologies. A sufficient condition for scaling is that each superdomain has at most $\log N_D$ sd-gateways; this condition is satisfied by realistic internetworks since most domain interconnections are "hierarchical connections" i.e. between backbones and regionals, between regionals and MANs, and so on.

In this section, we present an evaluation of the scaling properties of the superdomain hierarchy and the query protocol. To evaluate any inter-domain routing protocol, we need a model in which we can define internetwork topologies, policy/ToS constraints, inter-domain routing hierarchies, and evaluation measures (e.g. memory and time requirements). We have recently developed such a model [3]. We first describe our model, and then use it to evaluate our superdomain hierarchy. Our evaluation measures are the amount of memory required at the routers, and the amount of

---

[9] Even though the results in [10] were for intra-domain routing, it is easy to show that the analysis there holds for inter-domain routing as well.

time needed to construct a path.

7.1 Evaluation Model

We first describe our method of generating topologies and policy/ToS constraints. We then describe the evaluation measures.

Generating Internetwork Topologies

For our purposes, an internetwork *topology* is a directed graph where the nodes correspond to domains and the edges correspond to domain-level connections. However, an arbitrary graph will not do. The topology should have the characteristics of a real internetwork, like the Internet. That is, it should have backbones, regionals, MANS, LANS, etc.; there should be hierarchical connections, but some "non-hierarchical" connections should also be present.

For brevity, we refer to backbones as class 0 domains, regionals as class 1 domains, metropolitan-area domains and providers as class 2 domains, and campus and local-area domains as class 3 domains. A (strictly) hierarchical interconnection of domains means that class 0 domains are connected to each other, and for $i > 0$, class i domains are connected to class $i - 1$ domains. As mentioned above, we also want some "non-hierarchical" connections, i.e., domain-level edges between domains irrespective of their classes (e.g. from a campus domain to another campus domain or to a backbone domain).

In reality, domains span geographical regions and domain-level edges are often between domains that are geographically close (e.g. University of Maryland campus domain is connected to SURANET regional domain which are both in the east coast). We also want some edges that are between far domains. A class i domain usually spans a larger geographical region than a class $i + 1$ domain. To generate such interconnections, we associate a "region" attribute to each domain. The intention is that two domains with the same region are geographically close.

The *region* of a class i domain has the form $r_0.r_1.\cdots.r_i$, where the r_j's are integers. For example, the region of a class 3 domain can be 1.2.3.4. For brevity, we refer to the region of a class i domain as a class i region.

Note that regions have their own hierarchy which should not be confused with the superdomain hierarchy. Class 0 regions are the top level regions. We say that a class i region $r_0.r_1.\cdots.r_{i-1}.r_i$

is *contained* in the class $i-1$ region $r_0.r_1.\cdots.r_{i-1}$ (where $i > 0$). Containment is transitive. Thus region 1.2.3.4 is contained in regions 1.2.3, 1.2 and 1.
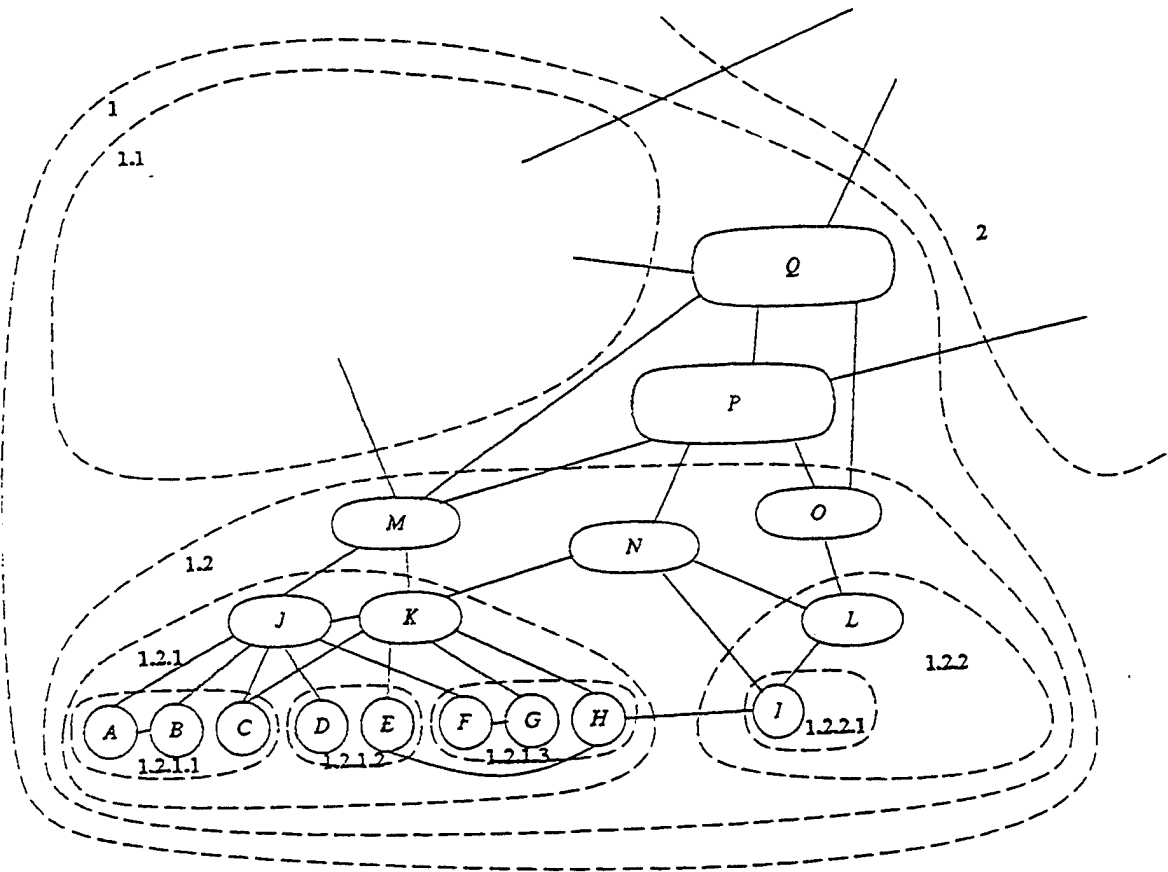
Figure 11: Regions

Given any pair of domains, we classify them as local, remote or far, based on their regions. Let X be a class i domain and Y a class j domain, and without loss of generality let $i \leq j$. X and Y are *local* if they are in the same class i region. For example in Figure 11, A is local to B, C, J, K, M, N, O, P, and Q. X and Y are *remote* if they are not in the same class i region but they are in the same class $i-1$ region, or if $i = 0$. For example in Figure 11, some of the domains A is remote to are D, E, F, and L. X and Y are *far* if they are not local or remote. For example in Figure 11, A is far to I.

We refer to a domain-level edge as *local* (*remote*, or *far*) if the two domains it connects are local

(remote, or far).

We use the following procedure to generate internetwork topologies:

- We first specify the number of domain classes, and the number of domains in each class.

- We next specify the regions. Note that the number of region classes equals the number of domain classes. We specify the number of class 0 regions. For each class $i > 0$, we specify a *branching factor*, which creates that many class i regions in each class $i - 1$ region. (That is, if there are two class 0 regions and the class 1 branching factor equals three, then there are six class 1 regions.)

- For each class i, we randomly map the class i domains into the class i regions. Note that several domains can be mapped to the same region, and some regions may have no domain mapped into them.

- For every class i and every class j, $j \geq i$, we specify the number of local, remote and far edges to be introduced between class i domains and class j domains. The end points of the edges are chosen randomly (within the specified constraints).

- We ensure that the internetwork topology is connected by ensuring that the subgraph of class 0 domains is connected, and each class i domain, for $i > 0$, is connected to a local class $i - 1$ domain.

- Each domain has one gateway. So all neighbors of a domain are connected via this gateway. This is for simplicity.

Choosing Policy/ToS Constraints

We chose a simple scheme to model policy/ToS constraints. Each domain is assigned a color: *green* or *red*. For each domain class, we specify the percentage of green domains in that class, and then randomly choose a color for each domain in that class.

A *valid route* from a source to a destination is one that does not visit any red intermediate domains; the source and destination domains are allowed to be red.

This simple scheme can model many realistic policy/ToS constraints, such as security constraints and bandwidth requirements. It cannot model some important kinds of constraints, such as delay bounds.

Computing Evaluation Measures

The evaluation measures of most interest for an inter-domain routing protocol are its memory, time and communication requirements. We postpone the precise definitions of the evaluation measures to the next subsection.

The only analysis method we have at present is to numerically compute the evaluation measures for a variety of source-destination pairs. Because we use internetwork topologies of large sizes, it is not feasible to compute for all possible source-destination pairs. We randomly choose a set of source-destination pairs that satisfy the following conditions: (1) the source and destination domains are different stub domains, and (2) there exists a valid path from the source domain to the destination domain in the internetwork topology. (Note that the straight-forward scheme would always find such a path.)

7.2 Application to Superdomain Query Protocol

We use the above model to evaluate our superdomain query protocol for several different superdomain hierarchies. For each hierarchy, we define a set of superdomain-ids and a parent-child relationship on them.

The first superdomain hierarchy scheme is referred to as *child-domains*. Each domain d (regardless of its class) is a level-1 superdomain, also identified as d. In addition, for each backbone d, we create a distinct level-4 superdomain referred to as d-4. For each regional d, we create a distinct level-3 superdomain d-3 and make it a child of a randomly chosen level-4 superdomain e-4 such that d and e are local and connected. For each MAN d, we create a distinct level-2 superdomain d-2 and make it a child of a randomly chosen level-3 superdomain e-3 such that d and e are local and connected. Please see Figure 12.

We next describe how the level-1 superdomains (i.e. the domains) are placed in the hierarchy. A backbone d is placed in, i.e. as a child of, d-4. A regional d is placed in d-3. A MAN d is placed in d-2. A stub d is placed in e-2 such that d and e are local and connected. Please see Figure 12.

The second superdomain hierarchy scheme is referred to as *sibling-domains*. It is identical to *child-domains* except for the placement of level-1 superdomains corresponding to backbones, regionals and MANs. In *sibling-domains*, a backbone d is placed as a sibling of d-4. A regional d is placed as a sibling of d-3. A MAN d is placed as a sibling of d-2. Please see Figure 13.

196
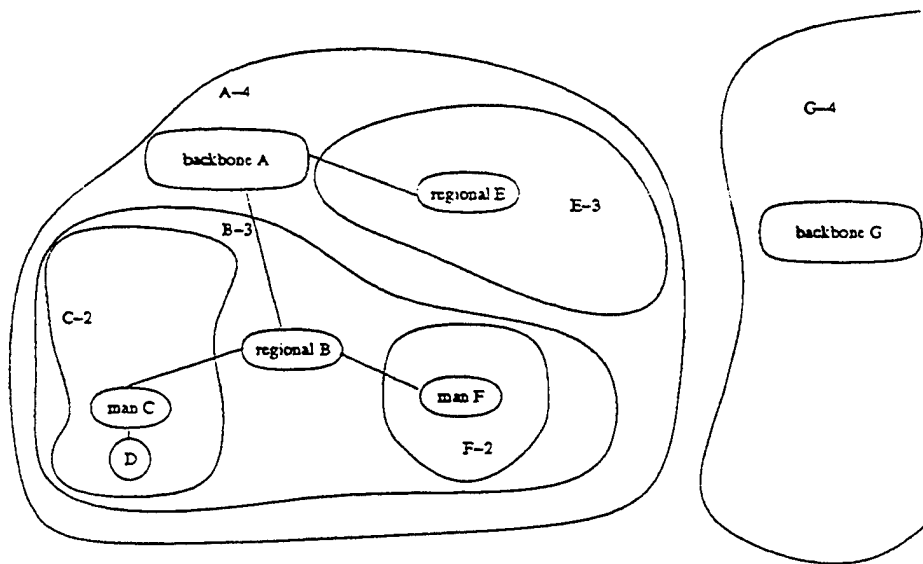
Figure 12: *child-domains*
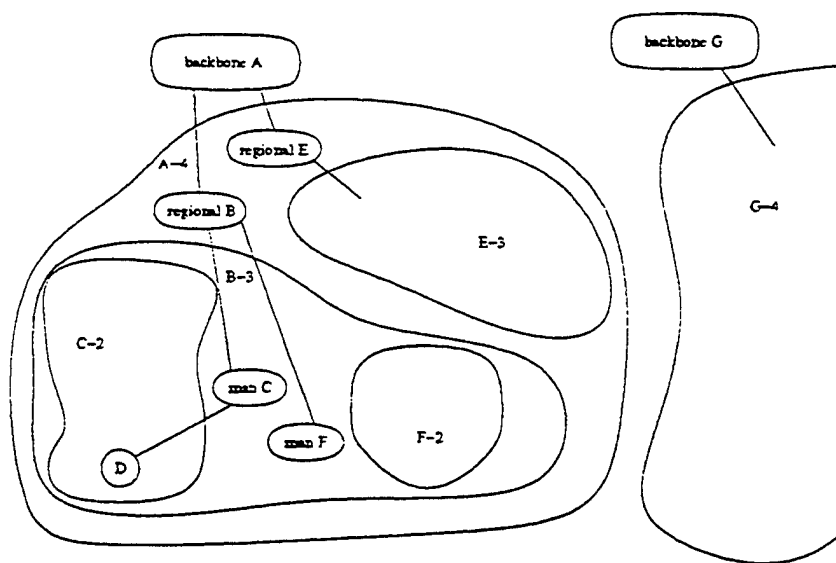
Figure 13: *sibling-domains*

The third superdomain hierarchy scheme is referred to as *leaf-domains*. It is identical to *child-domains* except for the placement of level-1 superdomains corresponding to backbones and regionals.

In *leaf-domains*, backbones and regionals are placed in some level-2 superdomain, as follows. A regional d, if superdomain d-3 has a child superdomain e-2, is placed in e-2. Otherwise, a new level-2 superdomain d-2 is created and placed in d-3. d is placed in d-2. A backbone d, if superdomain d-4 has a child superdomain f-3, is placed in the level-2 superdomain containing the regional f. Otherwise, a new level-3 superdomain d-3 is created and placed in d-4, a new level-2 superdomain d-2 is created and placed in d-3. d is placed in d-2. Please see Figure 14.

Note that in *leaf-domains*, all level-1 superdomains are placed under level-2 superdomains. Whereas other schemes allow some level-1 superdomains to be placed under higher level superdomains.
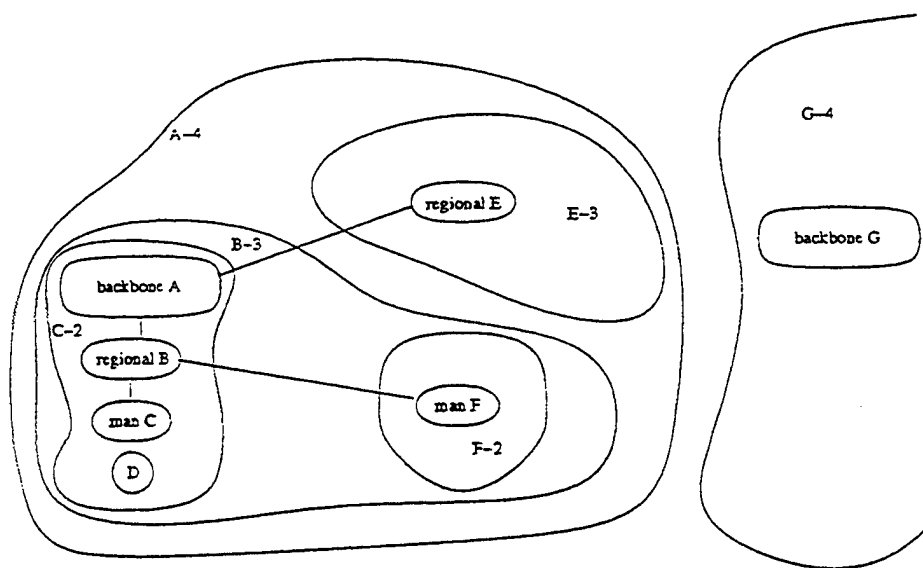
Figure 14: *leaf-domains*

The fourth superdomain hierarchy scheme is referred to as *regions*. In this scheme, the superdomain hierarchy corresponds exactly to the region hierarchy used to generate the internetwork topology. That is, for a class 1 region x there is a distinct level 5 (top level) superdomain x-5. For a class 2 region $x.y$ there is a distinct level 4 superdomain $x.y$-4 placed under level 5 superdomain x-5, and so on. Each domain is placed under the superdomain of its region. Please see Figure 11.

Results for Internetwork 1

The parameters of the first internetwork topology, referred to as Internetwork 1, are shown in Table 1.

Class i	No. of Domains	No. of Regions[10]	% of Green Domains	Edges between Classes i and j			
				Class j	Local	Remote	Far
0	10	4	0.80	0	8	6	0
1	100	16	0.75	0	190	20	0
				1	26	5	0
2	1000	64	0.70	0	100	0	0
				1	1060	40	0
				2	200	40	0
3	10000	256	0.20	0	100	0	0
				1	100	0	0
				2	10100	50	0
				3	50	50	50

Table 1: Parameters of Internetwork 1.

Our evaluation measures were computed for a (randomly chosen but fixed) set of 100,000 source-destination pairs. For a source-destination pair, we refer to the length of the shortest valid path in the internetwork topology as the *shortest-path length*, or *spl* in short. The minimum *spl* of these pairs was 2, the maximum *spl* was 15, and the average *spl* was 6.84.

For each source-destination pair, the set of candidate paths is examined in shortest-first order until either a valid path was found or the set was exhausted and no valid paths were found. For each candidate path, RequestIView messages are sent to all candidate superdomains on this path in parallel. All ReplyIView messages are received in time proportional to the round-trip time to the farthest of these superdomains. Hence, total time requirement is proportional to the number of candidate paths queried multiplied by the round-trip time to the farthest superdomain in these paths. Let *msgsize* denote the sum of average RequestIView message size and average

---

[10]Branching factor is 4 for all region classes.

Scheme	No query needed	Candidate Paths	Candidate Superdomains
child-domains	220	3.31/13	7.35/38
sibling-domains	220	3/10	6.17/22
leaf-domains	219	6.31/24	15.94/66
regions	544	3.70/12	7.79/30

Table 2: Queries for Internetwork 1.

`ReplyIView` message size. The number of candidate superdomains queried times *msgsize* indicates the communication capacity required to ship the `RequestIView` and `ReplyIView` messages.

Table 2 lists for each superdomain scheme the average and maximum number of candidate paths and candidate superdomains queried. As apparent from the table, *sibling-domains* is superior to other schemes and *leaf-domains* is much worse than the rest. This is because in *leaf-domains*, even if only one domain d in a superdomain U is actually going to be crossed, all descendants of U containing d may need to be queried to obtain a valid path (e.g. to cross backbone A in Figure 14, it may be necessary to query for superdomain A-4, then B-3, then C-2).

Scheme	Initial view size		Merged view size	
	in sd-gateways	in superdomains	in sd-gateways	in superdomains
child-domains	964/1006	42/60	1089/1282	100/298
sibling-domains	1167/1269	70/99	1470/2190	148/337
leaf-domains	963/1006	40/60	1108/1322	130/411
regions	492/715	85/163	1042/2687	158/369

Table 3: View sizes for Internetwork 1.

Table 3 lists for each superdomain scheme the average and maximum of the initial view size and of the merged view size. The initial view size indicates the memory requirement at a router without using the query protocol (i.e. assuming the initial view has a valid path). The merged view size indicates the memory requirement at a router during the query protocol (after finding a valid

path). The memory requirement at a router is O(view size in number of sd-gateways $\times E_G$) where E_G is the average number of edges of an sd-gateway. Note that the source does not need to store information about red and non-transit domains in the merged views (other than the ones already in the initial view). The numbers for the merged view sizes in Table 3 take advantage of this.

As apparent from the table, *leaf-domains*, *child-domains* and *regions* scale better than *sibling-domains*. There are two reasons for this. First, placing a backbone (regional or MAN) domain *d* as a sibling to *d-4* (*d-3* or *d-2*) doubles the number of level 4 (3 or 2) superdomains in the views of routers. Second, since these domains have many edges to the domains in their associated superdomains, the end points of each of these edges become sd-gateways of the associated superdomains. Note that *regions* scales much superior to the other schemes in the initial view size. This is because most edges are local (i.e. contained within regions), thus contained completely in superdomains. Hence, their end points are not sd-gateways.

Overall, the *child-domains* and *regions* schemes scale best in space, time and communication requirements. We have repeated the above evaluations for two other internetworks and obtained similar conclusions. The results are in Appendix A.

8 Related Work

In this section, we survey recently proposed inter-domain routing protocols that support ToS and policy routing for large internetworks.

Nimrod [6] and IDPR [16] use the link-state approach with domain-level source routing to enforce policy and ToS constraints and superdomains to solve scaling problem. Nimrod is still in a design stage. Both protocols suffer from loss of policy and ToS information as mentioned in the introduction. A query protocol for Nimrod is being developed to obtain more detailed policy, ToS and topology information.

BGP [12] and IDRP [14] are based on a *path-vector* approach [15]. Here, for each destination domain a router maintains a set of paths, one through each of its neighbor routers. ToS and policy information is attached to these paths. Each router requires $O(N_D \times N_D \times E_R)$ space, where N_D is the average number of neighbor domains for a domain and N_R is the number of routers in the internetwork. For each destination, a router exchanges its best valid path with its neighbor routers. However, a path-vector algorithm may not find a valid path from a source to the destination even

if such a route exists [16][11] (i.e. detailed ToS and policy information may be lost). By exchanging k paths to each destination, the probability of detecting a valid path for each source can be increased. But to guarantee detection, either all possible paths should be exchanged (exponential number of paths in the worst case) or source policies should be made public and routers should take this into account when exchanging routes. However, this fix increases space and communication requirements drastically. .

IDRP [14] uses superdomains to solve the scaling problem. It exchanges all paths between neighbor routers subject to the following constraint: a router does not inform a neighbor router of a route if usage of the route by the neighbor would violate some superdomain's constraint on the route. IDRP also suffers from loss of ToS and policy information. To overcome this problem, it uses overlapping superdomains: that is, a domain and superdomain can be in more than one parent superdomain. If a valid path over a domain can not be discovered because the constraints of a parent superdomain are violated, the same path may be discovered through another parent superdomain whose constraints are not violated. However, handling ToS and policy constraints in general requires more and more combinations of overlapping superdomains, resulting in more storage requirement.

Reference [9] combines the benefits of path-vector approach and link-state approach by having two modes: An NR mode, which is an extension of IDRP and is used for the most common ToS and policy constraints; and a SDR mode, which is like IDPR and is used for less frequent ToS and policy requests. This study does not address the scalability of the SDR mode. Ongoing work by this group considers a new SDR mode which is not based on IDPR.

Reference [19] suggests the use of multiple addresses for each node, one for each ToS and Policy. This scheme does not scale up. In fact, it increases the storage requirement, since a router maintains a route for each destination address, and there are more addresses with this scheme.

The landmark hierarchy [18, 17] is another approach for solving scaling problem. Here, each router is a landmark with a radius, and routers which are at most radius away from the landmark maintain a route for it. Landmarks are organized hierarchically, such that radius of a landmark increases with its level, and the radii of top level landmarks include all routers. Addressing and

---

[11] For example, suppose a router u has two paths $P1$ and $P2$ to the destination. Let u have a router neighbor v, which is in another domain. u chooses and informs v of one of the paths, say $P1$. But $P1$ may violate source policies of v's domain, and $P2$ may be a valid path for v.

packet forwarding schemes are introduced. Link-state algorithms can not be used with the landmark hierarchy, and a thorough study of enforcing ToS and policy constraints with this hierarchy has not been done.

In [1], we provided an alternative solution to loss of policy and ToS information that is perhaps more faithful to the original superdomain hierarchy. To handle superdomain-level source routing and topology changes, we augmented each superdomain-level edge (U, V) with the address of an "exit" domain u in U and an "entry" domain v in V. To obtain internal views, we added for each visible superdomain U the edges from U to domains outside the parent of U. Surprisingly, this approach and the gateway-level view approach have the same memory and communication requirements. However, the first approach results in much more complicated protocols.

Reference [2] presents interdomain routing protocols based on a new kind of hierarchy, referred to as the viewserver hierarchy. This approach also scales well to large internetworks and does not lose detail ToS and policy information. Here, special routers called viewservers maintain the view of domains in a surrounding precinct. Viewservers are organized hierarchically such that for each viewserver, there is a domain of a lower level viewserver in its view, and views of top level viewservers include domains of other top level viewservers. Appropriate addressing and route discovery schemes are introduced.

9   Conclusion

We presented a hierarchical inter-domain routing protocol which satisfies policy and ToS constraints, adapts to dynamic topology changes including failures that partition domains, and scales well to large number of domains.

Our protocol achieves scaling in space requirement by using superdomains. Our protocol maintains superdomain-level views with sd-gateways and handles topology changes by using a link-state view update protocol. It achieves scaling in communication requirement by flooding topology changes affecting a superdomain U over U's parent superdomain.

Our protocol does not lose detail in ToS, policy and topology information. It stores both a strong set of constraints and a weak set of constraints for each visible superdomain. If the weak constraints but not the strong constraints of a superdomain U are satisfied (i.e. the aggregation has resulted in loss of detail in ToS and policy information), then some paths through U may be valid.

Our protocol uses a query protocol to obtain a more detailed "internal" view of such superdomains, and searches again for a valid path. Our evaluation results indicate that the query protocol can be performed using 15% extra space.

One drawback of our protocols is that to obtain a source route, views are merged at or prior to the connection setup, thereby increasing the setup time. This drawback is not unique to our scheme [7, 16, 6, 9]. There are several ways to reduce this setup overhead. First, source routes to frequently used destinations can be cached. Second, the internal views of frequently queried superdomains can be cached at routers close to the source domain. Third, better heuristics to choose candidate paths and candidate superdomains to query can be developed.

We also described an evaluation model for inter-domain routing protocols. This model can be applied to other inter-domain routing protocols. We have not done so because precise definitions of the hierarchies in these protocols are not available. For example, to do a fair evaluation of IDPR[16], we need precise guidelines for how to group domains into superdomains, and how to choose between the strong and weak methods when defining policy/ToS constraints of superdomains. In fact, these protocols have not been evaluated in a way that we can compare them to the superdomain hierarchy.

References

[1] C. Alaettinoğlu and A. U. Shankar. Hierarchical Inter-Domain Routing Protocol with On-Demand ToS and Poicy Resolution. In Proc. *IEEE International Conference on Networking Protocols '93*, San Fransisco, California, October 1993.

[2] C. Alaettinoğlu and A. U. Shankar. Viewserver Hierarchy: A New Inter-Domain Routing Protocol and its Evaluation. Technical Report UMIACS-TR-93-98, CS-TR-3151, Department of Computer Science, University of Maryland, College Park, October 1993. Earlier version CS-TR-3033, February 1993.

[3] C. Alaettinoğlu and A. U. Shankar. Viewserver Hierarchy: A New Inter-Domain Routing Protocol. In Proc. *IEEE INFOCOM '94*, Toronto, Canada, June 1994. To appear.

[4] A. Bar-Noy and M. Gopal. Topology Distribution Cost vs. Efficient Routing in Large Networks. In Proc. *ACM SIGCOMM '90*, pages 242–252, Philadelphia, Pennsylvania, September 1990.

[5] L. Breslau and D. Estrin. Design of Inter–Administrative Domain Routing Protocols. In Proc. *ACM SIGCOMM '90*, pages 231–241, Philadelphia, Pennsylvania, September 1990.

[6] I. Castineyra, J. N. Chiappa, C. Lynn, R. Ramanathan, and M. Steenstrup. The Nimrod Routing Architecture. Internet Draft., March 1994. Available by anonymous ftp from research.ftp.com:pub/nimrod.

[7] D.D. Clark. Policy routing in Internet protocols. Request for Comment RFC-1102, Network Information Center, May 1989.

[8] D. Estrin. Policy requirements for inter Administrative Domain routing. Request for Comment RFC-1125, Network Information Center, November 1989.

[9] D. Estrin, Y. Rekhter, and S. Hotz. Scalable Inter-Domain Routing Architecture. In Proc. *ACM SIGCOMM '92*, pages 40–52, Baltimore, Maryland, August 1992.

[10] L. Kleinrock and F. Kamoun. Hierarchical Routing for Large Networks. *Computer Networks and ISDN Systems*, (1):155–174, 1977.

[11] B.M. Leiner. Policy issues in interconnecting networks. Request for Comment RFC-1124, Network Information Center, September 1989.

[12] K. Lougheed and Y. Rekhter. Border Gateway Protocol (BGP). Request for Comment RFC-1105, Network Information Center, June 1989.

[13] R. Perlman. Hierarchical Networks and Subnetwork Partition Problem. *Computer Networks and ISDN Systems*, 9:297–303, 1985.

[14] Y. Rekhter. Inter-Domain Routing Protocol (IDRP). Available from the author., 1992. T.J. Watson Research Center, IBM Corp.

[15] K. G. Shin and M. Chen. Performance Analysis of Distributed Routing Strategies Free of Ping-Pong-Type Looping. *IEEE Transactions on Computers*, 1987.

[16] M. Steenstrup. An Architecture for Inter-Domain Policy Routing. Request for Comment RFC-1478, Network Information Center, July 1993.

[17] P. F. Tsuchiya. The Landmark Hierarchy: Description and Analysis, The Landmark Routing: Architecture Algorithms and Issues. Technical Report MTR-87W00152, MTR-87W00174, The MITRE Corporation, McLean, Virginia, 1987.

[18] P. F. Tsuchiya. The Landmark Hierarchy:A New Hierarchy For Routing In Very Large Networks. In Proc. *ACM SIGCOMM '88*, August 1988.

[19] P. F. Tsuchiya. Efficient and Robust Policy Routing Using Multiple Hierarchical Addresses. In Proc. *ACM SIGCOMM '91*, pages 53–65, Zurich, Switzerland, September 1991.

A   Results for Other Internetworks

Results for Internetwork 2

The parameters of the second internetwork topology, referred to as Internetwork 2, are the same as the parameters of Internetwork 1 but a different seed is used for the random number generation.

Our evaluation measures were computed for a set of 100,000 source-destination pairs. The minimum *spl* of these pairs was 1, the maximum *spl* was 14, and the average *spl* was 7.13.

Table 5 and Table 4 shows the results. Similar conclusions as in the case of Internetwork 1 hold.

Results for Internetwork 3

The parameters of the third internetwork topology, referred to as Internetwork 3, are shown in Table 6. Internetwork 3 is more connected, more class 0, 1 and 2 domains are green, and more class 3 domains are red. Hence, we expect bigger view sizes in number of sd-gateways.

Scheme	No query needed	Candidate Paths	Candidate Superdomains
child-domains	205	4.52/20	10.22/47
sibling-domains	205	3.01/8	6.50/21
leaf-domains	205	8.80/32	21.34/82
regions	640	3.52/10	7.85/28

Table 4: Queries for Internetwork 2.

Scheme	Initial view size		Merged view size	
	in sd-gateways	in superdomains	in sd-gateways	in superdomains
child-domains	958/1012	43/60	1079/1269	118/306
sibling-domains	1153/1283	72/101	1480/2169	160/324
leaf-domains	956/1009	41/58	1095/1281	156/387
regions	624/1024	110/231	1356/3578	206/435

Table 5: View sizes for Internetwork 2.

Our evaluation measures were computed for a set of 100,000 source-destination pairs. The minimum spl of these pairs was 1, the maximum spl was 11, and the average spl was 5.95.

Table 8 and Table 7 shows the results. Similar conclusions as in the cases of Internetwork 1 and 2 hold.

---

[12]Branching factor is 4 for all domain classes.

Class i	No. of Domains	No. of Regions[12]	% of Green Domains	Class j	Local	Remote	Far
0	10	4	0.85	0	8	7	0
1	100	16	0.80	0	190	20	0
				1	50	20	0
2	1000	64	0.75	0	500	50	0
				1	1200	100	0
				2	200	40	0
3	10000	256	0.10	0	300	50	0
				1	250	100	0
				2	10250	150	50
				3	200	150	100

Table 6: Parameters of Internetwork 3.

Scheme	No query needed	Candidate Paths	Candidate Superdomains
child-domains	142	3.99/29	7.70/43
sibling-domains	142	2.95/10	5.39/22
leaf-domains	142	9.65/70	18.99/103
regions	676	3.47/17	6.25/21

Table 7: Queries for Internetwork 3.

Scheme	Initial view size		Merged view size	
	in sd-gateways	in superdomains	in sd-gateways	in superdomains
child-domains	2160/2239	43/60	2354/2647	107/348
sibling-domains	2365/2504	72/101	2606/3314	148/356
leaf-domains	2159/2236	41/58	2386/2645	160/648
regions	1107/1644	110/231	1850/3559	194/436

Table 8: View sizes for Internetwork 3.

```
Variables:

  View_x. Dynamic view of x.

  WView_x(d_address). Temporary view of x. d_address is the destination address.
      Used for merging internal views of superdomains to the view of x.

  PendingReq_x(d_address). Integer. d_address is the destination address.
      Number of outstanding request messages.

Events:
  Request_x(d_address)         {Executed when x wants a valid domain-level source route}
      allocate WView_x(d_address) := View_x;  allocate PendingReq_x(d_address) := 0;
      search_x(d_address);

where
  search_x(d_address)
      if there is a valid path to d_address in WView_x(d_address) then
          result := shortest valid path;
          deallocate WView_x(d_address), PendingReq_x(d_address);
          return result;
      else if there is a candidate path to d_address in WView_x(d_address) then
          Let cpath = (U_0:g_{0_0},...,U_0:g_{0_{n_0}}, U_1:g_{1_0},...,U_1:g_{1_{n_1}}, ··· ,U_m:g_{m_0},...,U_m:g_{m_{n_m}})
              be the shortest candidate path;
          for U_i in cpath such that U_i is candidate  do
              ReliableSend(RequestIView, U_i, g_{i_0}, address(x), d_address) to g_{i_0}
              PendingReq_x(d_address) := PendingReq_x(d_address) + 1;
      else
          deallocate WView_x(d_address), PendingReq_x(d_address);
          return failure;
        endif
      endif

  TimeOut_x(d_address)         {Executed after a time-out period and PendingReq_x(d_address) ≠ 0.}
      deallocate WView_x(d_address), PendingReq_x(d_address);
      return failure;
```

Figure 15: view-query protocol: State and events of a router x. (Figure continued on next page.)

```
Receive_x(RequestIView, sdid, x, s_address, d_address)
    ReliableSend(ReplyIView, sdid, x, IView_x(U), d_address) to s_address;


Receive_x(ReplyIView, sdid, gid, iview, d_address)
    if PendingReq_x(d_address) ≠ 0 then        {No time-out happened}
        PendingReq_x(d_address) := PendingReq_x(d_address) − 1;
            {merge internal view}
        delete ⟨sdid, *, *, *⟩ from WView_x;
        for ⟨child, scons, wcons, gateway-set⟩ in iview do
            if ¬∃⟨child, *, *, *⟩ ∈ WView_x then
                insert ⟨child, scons, wcons, gateway-set⟩ in WView_x;
            else
                for ⟨gid, ts, edge-set⟩ in gateway-set do
                    if ∃⟨gid, timestamp, *⟩ ∈ Gateways&Edges_x(child) ∧ ts > timestamp then
                        delete ⟨gid, *, *⟩ from Gateways&Edges_x(child);
                    endif;
                    if ¬∃⟨gid, *, *⟩ ∈ Gateways&Edges_x(child) then
                        insert ⟨gid, ts, edge-set⟩ to Gateways&Edges_x(child);
                    endif
            endif
        if PendingReq_x(d_address) = 0 then        {All pending replies are received}
            search_x(d_address);
        endif
    endif
```

Figure 15: view-query protocol: State and events of a router x. (cont.)

```
Constants:
  AdjLocalRouters_x. (⊆ NodeIds). Set of neighbor routers in x's domain.
  AdjForeignGateways_x. (⊆ NodeIds). Set of neighbor routers in other domains.
  Ancestor_i(x). (⊆ SuperDomainIds). ith ancestor of x.
Variables:
  View_x. Dynamic view of x.
  IntraDomainRT_x. Intra-domain routing table of x. Initially contains no entries.
  Clock_x : Integer. Clock of x.
Events:
  Receive_x(Update, sdid, gid, ts, edge-set) from sender
      if ∃⟨gid, timestamp, *⟩ ∈ Gateways&Edges_x(sdid) ∧ ts > timestamp then
          delete ⟨gid, *, *⟩ from Gateways&Edges_x(sdid);
      endif;
      if ¬∃⟨gid, *, *⟩ ∈ Gateways&Edges_x(sdid) then
          flood_x((Update, sdid, gid, ts, edge-set));
          insert ⟨gid, ts, edge-set⟩ to Gateways&Edges_x(sdid);
          update_parent_domains_x(level(sdid) + 1);
      endif

where
  update_parent_domains_x(startinglevel)
      for level := startinglevel to number of levels in the hierarchy  do
          sdid := Ancestor_level(x);
          if x ∈ Gateways(sdid) then
              edge-set := aggregate edges of sdid:x using View_x, IntraDomainRT_x and links of x;
              timestamp = Clock_x;
              flood_x((Update, sdid, x, timestamp, edge-set));
              delete ⟨x, *, *⟩ from Gateways&Edges_x(sdid);
              insert ⟨x, timestamp, edge-set⟩ to Gateways&Edges_x(sdid);
          endif

  Do_Update_x        {Executed periodically and upon a change in IntraDomainRT_x or links of x}
      update_parent_domains_x(1)

  Link_Recovery_x(y)        {⟨x, y⟩ is a link. Executed when ⟨x, y⟩ recovers.}
      for all ⟨sdid, *, *, *⟩ in View_x do
          if ∃i : Ancestor_i(y) = Ancestor_1(sdid) then
              for all ⟨gid, timestamp, edge-set⟩ in Gateways&Edges_x(sdid) do
                  Send((Update, sdid, gid, timestamp, edge-set)) to y;
          endif

  flood_x(packet)
      for all y ∈ AdjLocalRouters_x do
          Send(packet) to y;
      for all y ∈ AdjForeignGateways_x ∧ ∃i : Ancestor_i(y) = Ancestor_1(packet.sdid) do
          Send(packet) to y;
```

Figure 16: view-update protocol: State and events of a router x.

REPORT DOCUMENTATION PAGE

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 20, 1994	Technical

4. TITLE AND SUBTITLE

Hierarchical Inter-Domain Routing Protocol with On-Demand ToS and Policy Resolution

5. FUNDING NUMBERS

DASG-60-92-C-0055

6. AUTHOR(S)

Cengiz Alaettinoglu and A. Udaya Shankar

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Maryland
A. V. Williams Building
Department of Computer Science
College Park, MD 20742

8. PERFORMING ORGANIZATION REPORT NUMBER

CS-TR 3299
UMIACS-TR-94-73

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Phillips Labs
3550 Aberdeen Ave. SE
Kirtland AFB, NM 87117-5776

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Traditional inter-domain routing protocols based on superdomains maintain either "strong" or "weak" ToS and policy constraints for each visible superdomain. With strong constraints, a valid path may not be found even though one exists. With weak constraints, an invalid domain-level path may be treated as a valid path.

We present an inter-domain routing protocol based on superdomains, which always finds a valid path if one exists. Both strong and weak constraints are maintained for each visible superdomain. If the strong constraints of the superdomains on a path are satisfied, then the path is valid. If only the weak constraints are satisfied for some superdomains on the path, the source uses a query protocol to obtain a more detailed "internal" view of these superdomains, and searches again for a valid path. Our protocol handles topology changes, including node/link failures that partition superdomains. Evaluation results indicate our protocol scales well to large internetworks.

14. SUBJECT TERMS (Routing Protocols);(Computer Network Routing Protocols) (Computer-Communication Networks): Network Architecture and Design- packet networks; store and forward networks; (Computer Communication Networks:Network Protocols-protocol architecture

15. NUMBER OF PAGES
34

16. PRICE CODE
N/A

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited

Optimization in Non-Preemptive Scheduling for Aperiodic Tasks *

Shyh-In Hwang   Sheng-Tzong Cheng

Ashok K. Agrawala

Institute for Advanced Computer Studies

and

Systems Design and Analysis Group

Department of Computer Science

University of Maryland

College Park, MD 20742

Abstract

Real-time computer systems have become more and more important in many applications, such as robot control, flight control, and other mission-critical jobs. The correctness of the system depends on the temporal correctness as well as the functional correctness of the tasks. We propose a scheduling algorithm based on an analytic model. Our goal is to derive the optimal schedule for a given set of aperiodic tasks such that the number of rejected tasks is minimized, and then the finish time of the schedule is also minimized. The scheduling problem with a nonpreemptive discipline in a uniprocessor system is considered. We first show that if a total ordering is given, this can be done in $O(n^2)$ time by dynamic programming technique, where n is the size of the task set. When the restriction of the total ordering is released, it is known

to be NP-complete [3]. We discuss the super sequence [18] which has been shown to be useful in reducing the search space for testing the feasibility of a task set. By extending the idea and introducing the concept of conformation, the scheduling process can be divided into two phases: computing the pruned search space, and computing the optimal schedule for each sequence in the search space. While the complexity of the algorithm in the worst case remains exponential, our simulation results show that the cost is reasonable for the average case.

1   Introduction

In a hard real-time system, the computer is required to support the execution of applications in which the timing constraints of the tasks are specified. The correctness of the system depends on the temporal correctness as well as the functional correctness of the tasks. Failure to satisfy the timing constraints can incur fatal errors. Once a task is accepted by the system, the system should be able to finish it under the timing constraint of the task. A task T_i can be characterized as a triple of (r_i, c_i, d_i), representing the ready time, the computation time, and the deadline of the task, respectively. A task can not be started before its ready time. Once started, the task must use the processor for a consecutive period of c_i, and be finished by its deadline. The task set is represented as $\Gamma = \{T_1, T_2, ..., T_n\}$. A task set is *feasible* if there exists a *schedule* in which all the tasks in the task set can meet their timing constraints. Scheduling is a process of binding starting times to the tasks such that each task executes according to the schedule. A sequence $S = \langle T_1^S, T_2^S, ..., T_k^S \rangle$, where $k \le n$. T_i^S represents the ith task of the sequence S for any $1 \le i \le k$. A sequence specifies the order in which the tasks are executed. Without confusion, a schedule can be represented as a sequence. How to schedule the tasks so that the timing constraints are met is nontrivial. Many scheduling problems are known to be intractable [3] in that finding the optimal schedule requires large amounts of computations to be carried out.

The approaches adopted to date for scheduling algorithms can be generally classified into two categories. One approach is to assign priorities to tasks so that the tasks can be scheduled according to their priorities [1, 7, 8, 10, 12, 15, 14]. This approach is called *priority based scheduling*. The priority can be determined by deadline, execution time, resource requirement, laxity, period, or can be programmer-defined [4]. The other is *time based scheduling* approach [9, 13]. A time based scheduler generates as an output a *calendar* which specifies the time instants at which the tasks start and finish.

Generally speaking, scheduling for aperiodic task sets without preemption is NP-complete [3]. Due to the intractability, several search algorithms [11, 17, 19, 20] are proposed for computing optimal or suboptimal schedules. Analytic techniques may also be used for optimal scheduling. A dominance concept by Erschler *et al* [2] was proposed to reduce the search space for checking the feasibility of task sets. They explored the relations among the tasks

and determined the partial orderings of feasible schedules. Yuan and Agrawala [18] proposed decomposition methods to substantially reduce the search space based on the dominance concept. A task set is decomposed into subsets so that each subset can be scheduled independently. A super sequence is constructed to reduce search space further. Saksena and Agrawala [13] investigated the technique of temporal analysis serving as a pre-processing stage for scheduling. The idea is to modify the windows of two partially ordered tasks which are generated by the temporal relations so that more partial orderings of tasks may be generated recursively.

The time based model is employed by several real-time operating systems currently being developed, including MARUTI [5], MARS [6], and Spring [16]. In this paper, we study an analytic approach to optimal scheduling under the time based model. When complicated timing constraints and task interdependency are taken into consideration, the schedulability analysis of priority based scheduling algorithms becomes much more difficult. By analytic approach, we believe that the time based scheduling algorithm and analysis require reasonable amounts of computations to produce a feasible schedule.

The rest of this paper is organized as follows. In section 2, we describe how to compute the optimal schedule for a sequence. In section 3, releasing the restriction of total ordering on a sequence, we present the approach to computing the optimal schedule for a task set. Related theorems are also presented. In section 4, a simulation experiment is conducted to compare the performance of different algorithms. The last section is our conclusions.

2 Scheduling a Sequence

The *size* of a sequence (task set) is the number of tasks in the sequence(task set), and is denoted by $|S|$ ($|\Gamma|$). A sequence S is *feasible* if all tasks in S are executed in the order of the sequence and the timing constraints are satisfied. For convenience, we further define an *instance*, I, to be a sequence such that $|I| = |\Gamma|$. We denote the instance I by

$$I = \langle T_1^I, T_2^I, \ldots, T_n^I \rangle.$$

Notice that $\{\}$ is used to represent a task set, and $\langle \rangle$ a sequence. Let T_i and T_j be two tasks belonging to sequence S. If T_i is located before T_j in the sequence S, we say that
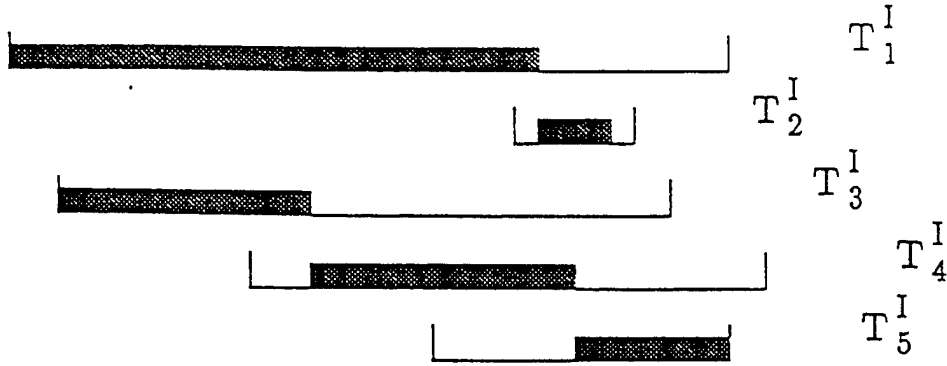
Figure 1: An instance $I = \langle T_1^I, T_2^I, \ldots, T_5^I \rangle$

$\langle T_i, T_j \rangle$ *conforms to* S. A sequence $S1$ *conforms to* a sequence $S2$, if, for any T_i and T_j, $\langle T_i, T_j \rangle$ conforming to $S1$ implies $\langle T_i, T_j \rangle$ conforming to $S2$. We use $\sigma(k)$ to represent the *optimal schedule* of $\langle T_1^I, T_2^I, \ldots, T_k^I \rangle$ in the sense that for any feasible sequence S conforming to $\langle T_1^I, T_2^I, \ldots, T_k^I \rangle$, either

$$|S| \quad < \quad |\sigma(k)|,$$

or

$$|S| \quad = \quad |\sigma(k)| \qquad and \qquad f_S \geq f_{\sigma(k)}, \tag{1}$$

where f_S and $f_{\sigma(k)}$ is the finish time of S and $\sigma(k)$ respectively. $\sigma(k)$ is thus the optimal schedule for the first k tasks of I. The optimal schedule for an instance I can thus be represented by $\sigma(n)$. For simplicity, let $u_k = |\sigma(k)|$. In this section, we will discuss the scheduling for an instance. However, the approach is generally applicable to any sequence.

2.1 Preliminary

We assume that $r_i + c_i \leq d_i$ holds for each task T_i in the task set Γ. At the first glance, one may attempt to compute $\sigma(k)$ based on $\sigma(k-1)$. However, with careful examination, we can find that merely computing $\sigma(k-1)$ does not suffice to compute $\sigma(k)$. This is illustrated by the example in Figure 1. From this example, we can obtain

$$\sigma(1) = \langle T_1^I \rangle$$

$$\sigma(2) = \langle T_1^I, T_2^I \rangle.$$

At the next step, $\sigma(2) \oplus \langle T_3^I \rangle$ is not feasible, where the operator $\oplus$ means concatenation of two sequences. One task must be rejected, which is T_3^I in this case. Hence, we got

$$\sigma(3) = \sigma(2) = \langle T_1^I, T_2^I \rangle.$$

A problem arises at the next step. $\sigma(3) \oplus \langle T_4^I \rangle$ is not feasible either. If we try to fix it by taking a task off this sequence, the result is

$$\sigma'(4) = \sigma(3) = \langle T_1^I, T_2^I \rangle.$$

However, the correct result should be

$$\sigma(4) = \langle T_3^I, T_4^I \rangle.$$

Although both $\sigma'(4)$ and $\sigma(4)$ are of the same size, the latter comes with a shorter finish time, which becomes significant at next step. We get

$$\sigma(n) = \sigma(5) = \sigma(4) \ominus \langle T_5^I \rangle = \langle T_3^I, T_4^I, T_5^I \rangle.$$

However, with $\sigma'(4)$, we would have

$$\sigma'(5) = \sigma'(4) = \langle T_1^I, T_2^I \rangle.$$

This example shows that merely computing $\sigma(k-1)$ does not suffice to compute $\sigma(k)$. When $\sigma(k-1)$ is obtained, it can not be predicted as to which tasks would be included in $\sigma(k)$. The approach has to be modified as follows.

2.2 Sequence-Scheduler Algorithm

We denote by $S(k,j)$ the sequence such that $S(k,j)$ conforms to $\langle T_1^I, T_2^I, \ldots, T_k^I \rangle$ and $|S(k,j)| = j$, where $j \leq |\sigma(k)|$. $S(k,j)$ represents any sequence of j tasks picked up from the first k tasks of S. We further define a sequence, denoted by $\sigma(k,j)$, to be the *optimal schedule with degree j* for $\langle T_1^I, T_2^I, \ldots, T_k^I \rangle$ in the sense that for any feasible sequence $S(k,j)$, we have

$$f_{\sigma(k,j)} \leq f_{S(k,j)}.$$

Notice that $\sigma(k)$ is an abbreviation of $\sigma(k, u_k)$. If a sequence $S(k,j)$ is not feasible, $f_{S(k,j)} = \infty$.

We would like to compute $\sigma(k,j)$ based on $\sigma(k-1,j')$, where $j' \leq j \leq |\sigma(k)|$. The basic idea is as follows. We know $\sigma(k,j)$ either contains T_k^I or not. If so, then the other $j-1$ tasks are picked up from the first $k-1$ tasks, and $\sigma(k-1, j-1)$ is one of the best choices. In this case, $\sigma(k,j) = \sigma(k-1, j-1) \oplus T_k^I$. If $\sigma(k,j)$ does not contain T_k^I, all of the j tasks should be picked up from the first $k-1$ tasks, and $\sigma(k-1,j)$ is one of the best choices. In this case, $\sigma(k,j) = \sigma(k-1,j)$. Whether taking T_k^I or not is determined by comparing which one of the sequences comes with a shorter finish time. Therefore, $\sigma(k,j)$ can be determined by $\sigma(k-1, j-1)$, and $\sigma(k-1,j)$. The computation of $\sigma(k-1,j)$ is in turn based on $\sigma(k-2, j-1)$, and $\sigma(k-2,j)$. In general, at each step k, we need to compute $\sigma(k,j)$ for $j = 1, 2, \ldots, |\sigma(k)|$. The algorithm *Sequence-Scheduler* in Figure 2 formalizes this idea. It is worth mentioning that the condition of the "while" statement in the algorithm is designed to let j increase from 1 through $|\sigma(k)|$. The correctness is verified in the next section.

2.3 Verification of Sequence-Scheduler Algorithm

The proof of the correctness of the algorithm along with some related lemmas are given below.

Lemma 1 Let $S1$ and $S2$ be two sequences such that $f_{S1} \leq f_{S2}$. If $S2 \oplus \langle T_x \rangle$ is feasible, then $f_{S1 \oplus \langle T_x \rangle} \leq f_{S2 \oplus \langle T_x \rangle}$.

Algorithm <u>Sequence-Scheduler</u>:

Input: an instance $I = \langle T_1^I, T_2^I, \ldots, T_n^I \rangle$
Output: the optimal schedule $\sigma(n) \equiv \sigma(n, u_n)$ for I

$\sigma(0,0) := \langle \rangle; \ u_0 = 0$
for $k := 1, 2, \ldots, n$
    $j := 1$
    while $(j \leq u_{k-1})$ or $((j = u_{k-1} + 1)$ and $(\sigma(k-1, u_{k-1}) \oplus \langle T_k^I \rangle$ is feasible$))$
        if $f_{\sigma(k-1,j-1) \oplus \langle T_k^I \rangle} < f_{\sigma(k-1,j)}$
           $\sigma(k,j) := \sigma(k-1, j-1) \oplus \langle T_k^I \rangle$
        else
          $\sigma(k,j) := \sigma(k-1, j)$
        endif
        $j := j + 1$
    endwhile
    $u_k := j - 1$
endfor

Figure 2: Sequence-Scheduler Algorithm

PROOF: This is straightforward via the following equations.

$$f_{S1\oplus\langle T_x\rangle} = max(f_{S1}, r_{T_x}) + c_{T_x}$$
$$\leq max(f_{S2}, r_{T_x}) + c_{T_x}$$
$$= f_{S2\oplus\langle T_x\rangle}$$

$\square$

Corollary 1 Let $S1$ and $S2$ be two sequences such that $f_{S1} \leq f_{S2}$. If $S2 \oplus S3$ is feasible, where $S3$ is another sequence, then $f_{S1\oplus S3} \leq f_{S2\oplus S3}$.

PROOF: This is a direct result of applying Lemma 1 repeatedly through the tasks in $S3$. $\square$

Lemma 2 $u_k = u_{k-1}$ or $u_k = u_{k-1} + 1$.

PROOF: It is obvious that $u_k \geq u_{k-1}$, where both u_k and u_{k-1} are integers. Let us assume that $u_k = u_{k-1} + \alpha$, and $\alpha \geq 2$. We are going to show that this assumption does not hold. We know $\sigma(k, u_k)$ either contains T_k^I or not. If $\sigma(k, u_k)$ contains T_k^I, we can represent $\sigma(k, u_k)$ as $S(k-1, u_k - 1) \oplus \langle T_k^I \rangle$, by picking up a proper sequence $S(k-1, u_k - 1)$. However, from the assumption above, we have $u_{k-1} = u_k - \alpha < u_k - 1 = |S(k-1, u_k - 1)|$. This contradicts the definition of u_{k-1}. On the other hand, if $\sigma(k, u_k)$ does not contain T_k^I, we can represent $\sigma(k, u_k)$ as $S(k-1, u_k)$. We have $u_{k-1} = u_k - \alpha < u_k$, which is a contradiction. The assumption thus does not hold. Therefore, we have $\alpha \leq 1$. $\square$

From this lemma, $\sigma(k, j)$ does exist for $j \leq u_{k-1}$. Furthermore, in the algorithm, $j = u_{k-1} + 1$ is tested to see if $u_k = u_{k-1} + 1$.

Theorem 1 For $k = 1, 2, \ldots, n$, and $j = 1, 2, \ldots, u_k$, if $f_{\sigma(k-1,j-1)\oplus\langle T_k^I\rangle} < f_{\sigma(k-1,j)}$, then $\sigma(k, j) = \sigma(k-1, j-1) \oplus \langle T_k^I \rangle$; otherwise, $\sigma(k, j) = \sigma(k-1, j)$.

PROOF: The proof is by induction on k. When $k = 1$, $I_1 = \langle T_1^I \rangle$. Since $u_1 \leq 1$ and $\langle T_1^I \rangle$ is feasible, $\sigma(1, 1) = \langle T_1^I \rangle$. It is easy to come up with the same result through this theorem. Thus holds the base case. We assume that we can compute $\sigma(k-1, j)$, for $j = 1, 2, \ldots, u_{k-1}$,

in the same way, and consider the case of k. Let us first bring forward three basic equations. Since $f_{\sigma(k-1,j-1)} \leq f_{S(k-1,j-1)}$, the following equation holds by Lemma 1

$$f_{\sigma(k-1,j-1)\oplus\langle T_k^I\rangle} \leq f_{S(k-1,j-1)\oplus\langle T_k^I\rangle}. \tag{2}$$

By induction hypothesis on $\sigma(k-1,j)$ such that $f_{\sigma(k-1,j)} \leq f_{S(k-1,j)}$, we have

$$if \quad f_{\sigma(k-1,j-1)\oplus\langle T_k^I\rangle} < f_{\sigma(k-1,j)}, \quad then \quad f_{\sigma(k-1,j-1)\oplus\langle T_k^I\rangle} < f_{S(k-1,j)}. \tag{3}$$

From Equation 2, we have

$$if \quad f_{\sigma(k-1,j)} \leq f_{\sigma(k-1,j-1)\oplus\langle T_k^I\rangle}, \quad then \quad f_{\sigma(k-1,j)} \leq f_{S(k-1,j-1)\oplus\langle T_k^I\rangle}. \tag{4}$$

From Lemma 2, we know either $u_k = u_{k-1}+1$ or $u_k = u_{k-1}$. The two cases are discussed below.

Case I: $u_k = u_{k-1}+1$. We first discuss the situation when $j = 1, 2, \ldots, u_k - 1$. We know that a feasible sequence $S(k,j)$ is either in the form of $S(k-1,j)$ or $S(k-1,j-1) \oplus \langle T_k^I\rangle$. If $f_{\sigma(k-1,j-1)\oplus\langle T_k^I\rangle} < f_{\sigma(k-1,j)}$, then $f_{\sigma(k-1,j-1)\oplus\langle T_k^I\rangle} < f_{S(k-1,j)}$ by Equation 3. Also we have $f_{\sigma(k-1,j-1)\oplus\langle T_k^I\rangle} \leq f_{S(k-1,j-1)\oplus\langle T_k^I\rangle}$ by Equation 2. This means $f_{\sigma(k-1,j-1)\oplus\langle T_k^I\rangle} \leq f_{S(k,j)}$ for any feasible sequence $S(k,j)$. Consequently, $\sigma(k,j) = \sigma(k-1,j-1) \oplus \langle T_k^I\rangle$, which justifies the theorem. On the other hand, if $f_{\sigma(k-1,j-1)\oplus\langle T_k^I\rangle} \geq f_{\sigma(k-1,j)}$, then $f_{\sigma(k-1,j)} \leq f_{S(k-1,j-1)\oplus\langle T_k^I\rangle}$ by Equation 4. In addition, $f_{\sigma(k-1,j)} \leq f_{S(k-1,j)}$ by induction hypothesis on $\sigma(k-1,j)$. So $f_{\sigma(k-1,j)} \leq f_{S(k,j)}$ for any feasible sequences $S(k,j)$. In this case, $\sigma(k,j) = \sigma(k-1,j)$, which justifies the theorem.

Then we discuss the situation when $j = u_k$. Since $u_k = u_{k-1}+1$, it is clear that $\langle T_k^I\rangle$ belongs to $\sigma(k)$; otherwise, we need to pick up $u_{k-1}+1$ tasks from I_{k-1} to make a feasible sequence, which violates the definition of u_{k-1}. Therefore, $\sigma(k,j)$ can be expressed as $S(k-1, u_{k-1}) \oplus \langle T_k^I\rangle$ by picking up a proper sequence $S(k-1, u_{k-1})$. Note that $u_{k-1} = j-1$ here. By Equation 2, we have $f_{\sigma(k-1,j-1)\oplus\langle T_k^I\rangle} \leq f_{S(k-1,j-1)\oplus\langle T_k^I\rangle}$, for any sequence $S(k-1,j-1) \oplus \langle T_k^I\rangle$. Thus, $\sigma(k,j) = \sigma(k-1,j-1) \oplus \langle T_k^I\rangle$. Now let us check the theorem. The sequence $\sigma(k-1,j) = \sigma(k-1, u_{k-1}+1)$ is not feasible; thus its finish time is ∞. The condition $f_{\sigma(k-1,j-1)\oplus\langle T_k^I\rangle} < f_{\sigma(k-1,j)}$ is satisfied. So $\sigma(k,j) = \sigma(k-1,j-1) \oplus \langle T_k^I\rangle$. This justifies the theorem.

Case II: $u_k = u_{k-1}$. The reasoning follows the discussion of the first part in Case I. $\square$

3 Scheduling a Task Set

In this section we discuss how to schedule a task set by using Sequence-Scheduler. The *optimal schedule, ρ,* of a task set is defined as follows: for any feasible sequence S consisting of tasks in Γ, we have either

$$|S| < |\rho|,$$

or

$$|S| = |\rho| \qquad and \qquad f_S \geq f_\rho.$$

For simplicity, we use optimal schedule to represent the optimal schedule of the task set, when there is no confusion. Note that the optimal schedule of the task set is the *best* one of the optimal schedules of all instances in the task set. Erschler *et al* [2] proposed the dominance concept to reduce the number of permutations that should be examined for the feasibility test of a task set. Yuan and Agrawala [18] proposed the super sequence to further reduce the search space for testing the feasibility of a task set. In this section, we show that for our optimization problem, the super sequence provides a valid and pruned search space. In other words, there exists one optimal schedule which conforms to an instance in the super sequence of the task set. Thus we may use Sequence-Scheduler to schedule for the instances extracted from the super sequence to derive the optimal schedule. There may exist more than one optimal schedule for a task set. Our interest is on how to derive one of them.

3.1 Super Sequence

Temporal relations between two tasks T_i and T_j are summarized in the following. They are illustrated by Figure 3.

- leading : $T_i \prec T_j$, if $r_i \leq r_j, d_i \leq d_j$, but both of the equalities do not hold at the same time.

- matching : $T_i \parallel T_j$, if $r_i = r_j, d_i = d_j$.

- containing : $T_i \sqcup T_j$, if $r_i < r_j, d_i > d_j$.
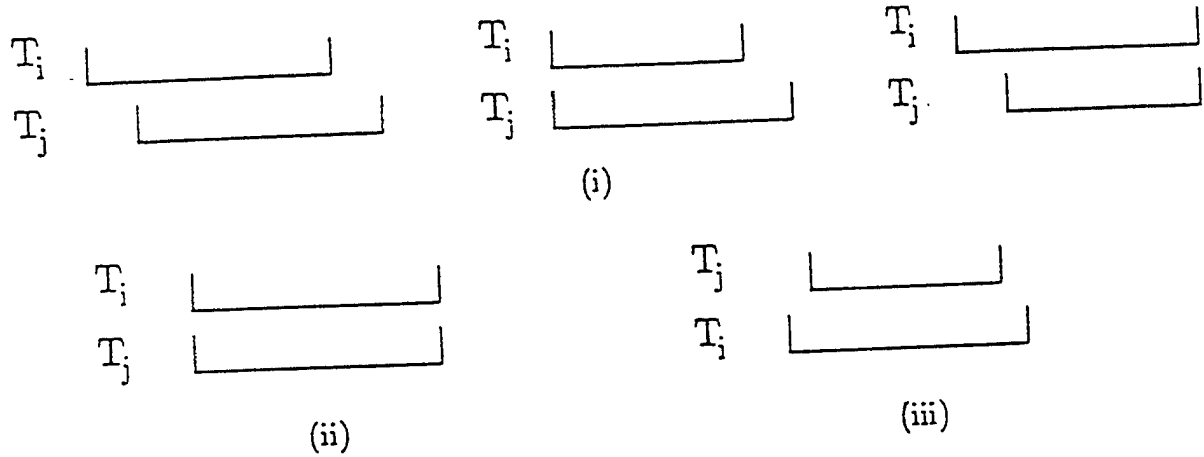
Figure 3: (i) $T_i \prec T_j$; (ii) $T_i \parallel T_j$; (iii) $T_i \sqcup T_j$

A task h is called a *top task* if there is no task contained by h. A task is called a *nontop task* if it contains at least one task. Assume that we have t top tasks in the task set, denoted by $h_1, h_2, \ldots, h_t$ respectively. Denote by M_k the set of tasks that contain the top task h_k, including h_k, and by $\overline{M_k}$ the set of tasks in the task set Γ that do not belong to M_k. We say that T_i is *weakly leading* to T_j, denoted by $T_i \lhd T_j$, if $T_i \prec T_j$ or $T_i \parallel T_j$. If $T_i \lhd T_j$ for all T_j belonging to S, then $T_i \lhd S$.

The dominance concept is originally developed by Erschler *et al* [2] to reduce the search space for testing the feasibility of a task set. The idea is extended with the super sequence proposed by Yuan and Agrawala [18]. An instance I *dominates* an instance I' iff:

$$I' \text{ feasible } \Rightarrow I \text{ feasible.}$$

It can be considered that I is a better candidate as a feasible schedule than I'. A *dominant instance* is an instance such that for each possible instance I of the task set, if I dominates the dominant instance, then the dominant instance dominates I. Thus the dominant instance can be considered as the best candidate of the feasible schedule. A set of instances is said to be a *dominant set*, if I does not belong to the dominant set, then there exists a dominant instance in the dominant set such that the dominant instance dominates I.

224

A super sequence Δ serves similarly as a dominant set in that there exists a dominant instance in the super sequence; and it is more appropriate for solving our problem. A super sequence is a sequence of tasks, where duplicates of tasks are allowed. The purpose is to extract instances from the super sequence for scheduling. The super sequence is constructed according to the dominant rules [2, 18] described below. Whenever a task satisfies one of the conditions specified by the rules, a duplicate of the task is inserted into the super sequence. Note that duplicates can only be generated for nontop tasks. The top tasks appear once and only once in the super sequence.

Rule R1: Let T_α and T_β be any two top tasks. If $T_\alpha \prec T_\beta$, then T_α is positioned before T_β. If $T_\alpha \parallel T_\beta$, the order of the two top tasks is determined arbitrarily.

A unique order of the top tasks can be thus determined for the super sequence. Let us denote the sequence composed of the top tasks by $H = \langle h_1, h_2, \ldots, h_t \rangle$. The rule implies that if T_α is positioned before T_β in the super sequence, then $T_\alpha \lhd T_\beta$. So $h_1 \lhd h_2 \lhd \ldots \lhd h_t$.

Rule R2:
(1) A nontop task can be positioned before the first top task h_1 only when it contains h_1.
(2) A nontop task can be positioned after the last top task h_t only when it contains h_t.
(3) A nontop task can be positioned between h_k and h_{k+1} only when it contains h_k or h_{k+1}.

The t top tasks delimit the super sequence into $t + 1$ regions by rule $R1$. Now we have $t + 1$ subsets of nontop tasks separated by the t top tasks by rule $R2$. Generally speaking, a nontop task has more than one possible location. Denote the kth subset by A_k, which is between top tasks h_k and h_{k+1}. From rule $R2$, it can be deduced that

$$A_k = B_{k,\overline{k+1}} \cup B_{k,k+1} \cup B_{\overline{k},k+1}, \text{ where} \tag{5}$$

$$B_{k,\overline{k+1}} = M_k \cap \overline{M_{k+1}}, \; B_{k,k+1} = M_k \cap M_{k+1}, \; B_{\overline{k},k+1} = \overline{M_k} \cap M_{k+1}.$$

Next rule is to specify the order of the tasks within each subset.

Rule R3: In each subset A_k, for $k = 0, 1, \ldots, n$,

(1) the tasks in $B_{k,\overline{k+1}}$ are ordered according to their deadlines, and tasks with the same deadlines are ordered arbitrarily,

(2) the tasks in $B_{k,k+1}$ are ordered arbitrarily,

(3) the tasks in $B_{\overline{k},k+1}$ are ordered according to their ready times, and tasks with the same ready times are ordered arbitrarily,

(4) the tasks in $B_{k,\overline{k+1}}$ are positioned before those in $B_{k,k+1}$, which in turn are positioned before those in $B_{\overline{k},k+1}$.

Now we are ready to construct the super sequence with these three rules. Top tasks are first picked out and ordered, forming $t + 1$ regions. In each region, there is a subsequence of nontop tasks. An instance extracted out of the super sequence is one that conforms to the super sequence without duplication of tasks. Let q be the number of top tasks that a nontop task contains. The number of possible regions the nontop task can fall into is $q + 1$. The number of instances in the super sequence thus sums up to

$$N = \prod_{q=1}^{q=t} (q + 1)^{n_q},$$

where n_q is the number of nontop tasks which contains q top tasks. Compared with an exhaustive search which takes up to $n!$ instances (permutations) into account, the super sequence generally leads to a smaller set. Notice that it takes $O(n)$ time to check if an instance is feasible. Hence, the time complexity of the feasibility test for the task set is $O(N * n)$.

3.2 Leading Theorem

The super sequence is not only useful in testing the feasibility of a task set; we will show that it is also useful in reducing the number of instances to be examined in order to obtain the optimal schedule of a task set. We will show that there exists at least one optimal schedule which conforms to an instance in the super sequence Δ. Hence, it suffices to check through Δ to obtain the optimal schedule of Γ.

226

It is worth attention that the top tasks in ρ may *not* be the same top tasks of Γ. This arises because some of the top tasks of Γ may be rejected, introducing new top tasks in ρ. Before proceeding to verify the rules for the super sequence, we will first introduce the *Leading Theorem*. It serves as the base for further analysis in the Dominance Theorem and Conformation Theorem to be described later. The Leading Theorem tells that under certain condition we can adjust the order of tasks to satisfy the Weakly Leading Condition to be defined below and do not introduce a schedule with greater finish time.

Assume that S is a feasible sequence, with $L_{pre}, L,$ and L_{post} subsequences of S such that

$$S = L_{pre} \oplus L \oplus L_{post}.$$

Let us denote L by

$$L = \langle T_\beta, T_{x_1}, T_{x_2}, \ldots, T_{x_\omega}, T_\alpha \rangle,$$

where $\omega \geq 0$. A frame F is defined to be a time interval characterized by a beginning time b_F, and an ending time e_F. We say that F is a frame corresponding to L, if $b_F = s_\beta$, and $e_F = f_\alpha$, where s_β is the starting time of T_β, and f_α is the finish time of T_α.

Theorem 2 (Leading Theorem) Assume that $S = L_{pre} \oplus L \oplus L_{post}$ is a feasible sequence, where $L = \langle T_\beta, T_{x_1}, T_{x_2}, \ldots, T_{x_\omega}, T_\alpha \rangle$. Let F be a frame corresponding to L. If $T_\alpha \prec T_\beta$, and there does not exist a task $T_{x_i}, 1 \leq i \leq \omega$, such that $F \sqcup T_{x_i}$, then there exists a sequence $\bar{L}$ which is a permutation of L such that
(i) $\langle T_\alpha, T_\beta \rangle$ conforms to $\bar{L}$, and
(ii) $f_{L_{pre} \oplus \bar{L} \oplus L_{post}} \leq f_{L_{pre} \oplus L \oplus L_{post}}$.

Before we can proceed to prove the theorem, the following definition is useful.

Weakly Leading Condition: a sequence $S = \langle T_1^S, T_2^S, \ldots, T_k^S \rangle$ satisfies Weakly Leading Condition if $T_1^S \lhd T_2^S \lhd \ldots \lhd T_k^S$.

Lemma 3 Let S be a sequence satisfying Weakly Leading Condition. If $\langle T_i, T_j \rangle$ conforms to S and $T_i \parallel T_j$, then all tasks located between T_i and T_j in S must match T_i and T_j.

PROOF: For any task T_x located between T_i and T_j, according to the definition of Weakly Leading Condition, we have $r_i \leq r_x \leq r_j$ and $d_i \leq d_x \leq d_j$. Since $T_i \parallel T_j$, $r_i = r_j$ and $d_i = d_j$. Therefore, we have $r_i = r_x = r_j$ and $d_i = d_x = d_j$. So, T_x matches T_i and T_j. $\square$

To obtain $\tilde{L}$, let us modify the tasks in L in the following way. If the ready time of a task is less than b_F, then its ready time is set to b_F. If the deadline of a task is greater than e_F, then its deadline is set to e_F. The computation times remain unchanged. Let L' be a sequence consisting of the modified tasks with the same order of L, i.e.,

$$L' = \langle T'_\beta, T'_{x_1}, T'_{x_2}, \ldots, T'_{x_\omega}, T'_\alpha \rangle.$$

Since $T_\alpha \prec T_\beta$, $d_\beta \geq d_\alpha \geq f_\alpha = e_F$. So $d'_\beta = d'_\alpha = e_F$. Also $r_\alpha \leq r_\beta \leq s_\beta = b_F$, so $r'_\alpha = r'_\beta = b_F$. This is illustrated in Fig 4 (ii).

Note that swapping T'_β and T'_α in the sequence does not result in a feasible sequence in this example. It is essential that we adjust the order of the tasks located between them. Let $\tilde{L}'$ be a sequence which is a permutation of L' and satisfies the Weakly Leading Condition, and to which $\langle T'_\alpha, T'_\beta \rangle$ conforms. Furthermore, $\tilde{L}'$ satisfies an even stronger condition. If $T_i^{\tilde{L}'}$ is positioned before $T_j^{\tilde{L}'}$ in $\tilde{L}'$, then $T_i^{\tilde{L}'} \lhd T_j^{\tilde{L}'}$; if, furthermore, $T_i^{\tilde{L}'} \parallel T_j^{\tilde{L}'}$, the corresponding tasks $T_i^{\tilde{L}'}$ and $T_j^{\tilde{L}'}$ satisfies that $T_i^{\tilde{L}'} \lhd T_j^{\tilde{L}'}$. The idea of such arrangement is that when interchanging T'_β and T'_α, we do not produce a new *reversed pair* like them. By reversed pair we mean for example $T_\alpha \prec T_\beta$ but T_β is positioned before T_α in the sequence. So, if $\langle T_i^{\tilde{L}'}, T_j^{\tilde{L}'} \rangle$ conforms to $\tilde{L}'$, the corresponding tasks satisfies the condition that either $T_i^{\tilde{L}'} \prec T_j^{\tilde{L}'}$ or $T_i^{\tilde{L}'} \sqcup T_j^{\tilde{L}'}$ or $T_j^{\tilde{L}'} \sqcup T_i^{\tilde{L}'}$. One possibility of $\tilde{L}'$ is illustrated in Fig 4 (iii), or

$$\tilde{L}' = \langle T'_{x_2}, T'_\alpha, T'_{x_3}, T'_\beta, T'_{x_1}, T'_{x_4} \rangle.$$

The existence of such a sequence is proved later. Finally, $\tilde{L}$ can be a sequence with the same order of $\tilde{L}'$, but the ready times and deadlines of the tasks are recovered to their original settings. This is illustrated in Fig 4 (iv). The figures give the rough idea about how the adjustment of task order can be made to satisfy the conditions described in the Leading Theorem. Here below is the proof of the Leading Theorem.

228
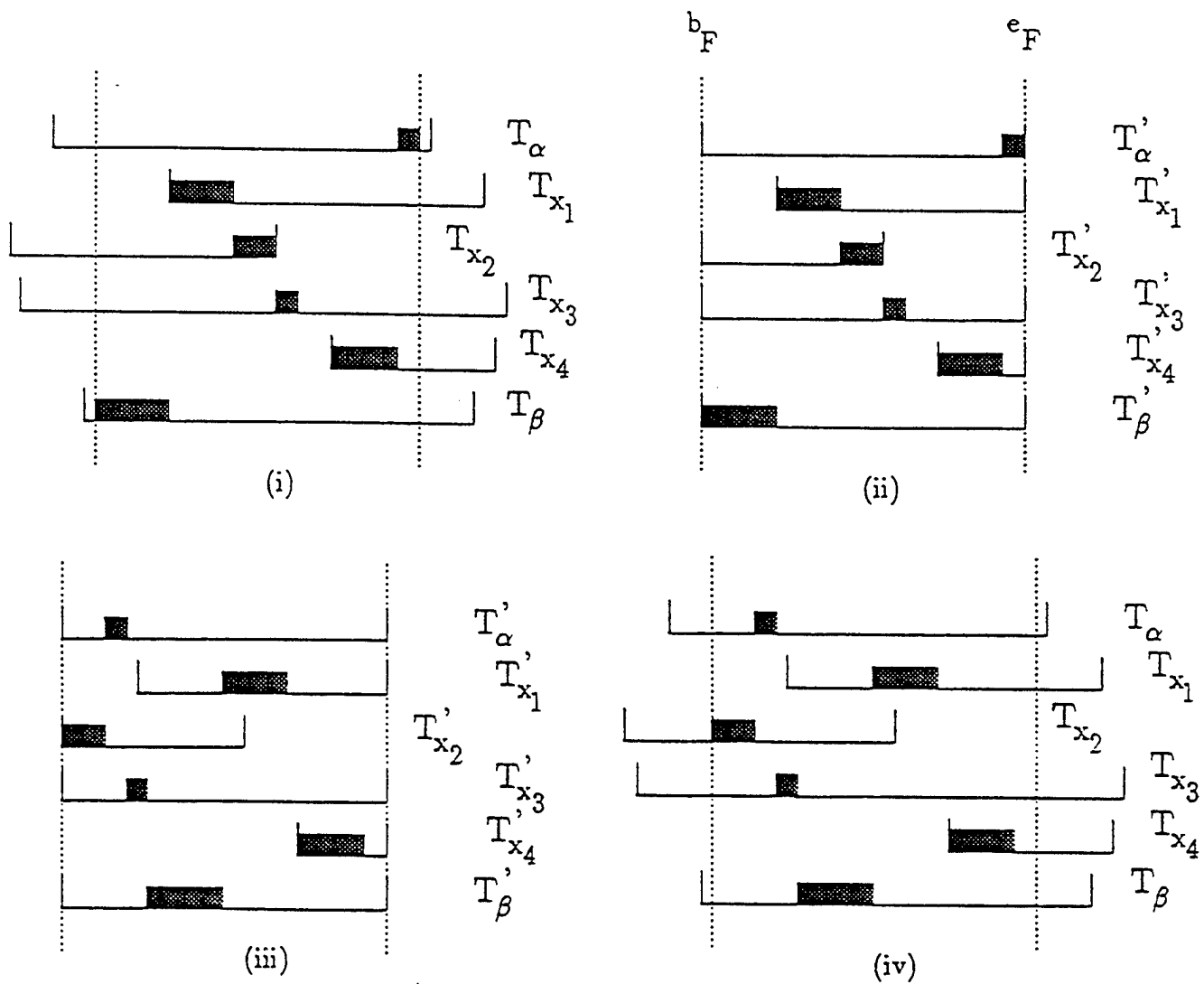
Figure 4: (i) $L = \langle T_\beta, T_{x_1}, T_{x_2}, T_{x_3}, T_{x_4}, T_\alpha \rangle$ (ii) $L' = \langle T'_\beta, T'_{x_1}, T'_{x_2}, T'_{x_3}, T'_{x_4}, T'_\alpha \rangle$ (iii) $\tilde{L}' = \langle T'_{x_2}, T'_\alpha, T'_{x_3}, T'_\beta, T'_{x_1}, T'_{x_4} \rangle$ (iv) $\tilde{L} = \langle T_{x_2}, T_\alpha, T_{x_3}, T_\beta, T_{x_1}, T_{x_4} \rangle$

229

PROOF (of Leading Theorem): We would first show the existence of $\tilde{L}$. The modification of ready times and deadlines of the tasks for L' is done in such a way that their started times are not affected. In addition, their computation times remain the same. It is clear that L' is feasible, and

$$f_{L_{pre}\oplus L'} = f_{L_{pre}\oplus L}.$$

We can obtain $\tilde{L}'$ in the following way. At the first step, the first task $T_1^{\tilde{L}'}$ of $\tilde{L}'$ is the task in L' such that, for any task T_x belonging to L', $T_1^{\tilde{L}'} \lhd T_x$. Such a task $T_1^{\tilde{L}'}$ exists because there are no containing relations among the modified tasks, and ties can be broken arbitrarily. $T_1^{\tilde{L}'}$ is exchanged with the task located just left to it in L'. Continue the exchanging process until $T_1^{\tilde{L}'}$ occupies the first location in the sequence. At the second step, the second task $T_2^{\tilde{L}'}$ of $\tilde{L}'$ is the task in L' such that, for any task T_x belonging to L' except $T_1^{\tilde{L}'}$, $T_2^{\tilde{L}'} \lhd T_x$. Exchange $T_2^{\tilde{L}'}$ with its left neighbor task consecutively until it occupies the second location in the sequence. At the ith step, the ith task $T_i^{\tilde{L}'}$ of $\tilde{L}'$ is the task in L' such that, for any task T_x belonging to L' except $T_1^{\tilde{L}'}$ through $T_{i-1}^{\tilde{L}'}$, $T_i^{\tilde{L}'} \lhd T_x$. Exchange $T_i^{\tilde{L}'}$ with its left neighbor task consecutively until it occupies the ith location in the sequence. We keep performing this operation until we finally obtain $\tilde{L}'$. Insertion of $T_i^{\tilde{L}'}$, $1 \le i \le |\tilde{L}'|$, into the ith position of the sequence by consecutive swapping is possible because $T_i^{\tilde{L}'} \lhd T_x$ for all T_x not belonging to $\langle T_1^{\tilde{L}'} \ldots T_{i-1}^{\tilde{L}'} \rangle$. In a word, the adjustment is possible because there are no containing relations among the modified tasks, and hence there exists a total ordering of the modified tasks by the Weakly Leading Condition. The resultant $\tilde{L}'$ is existent and is a sequence satisfying the Weakly Leading Condition.

There is a chance that $\langle T_\beta, T_\alpha \rangle$ conforms to $\tilde{L}'$. By Lemma 3, all tasks located between T_β and T_α must match each other. Hence the order of these tasks does not make any difference. We can thus exchange the position of T_β and T_α, which makes $\langle T_\beta, T_\alpha \rangle$ conform to $\tilde{L}'$

During the process of adjusting the position of $T_i^{\tilde{L}'}$, $1 \le i \le |\tilde{L}'|$, $T_i^{\tilde{L}'}$ leads to or matches any task in the sequence except $\langle T_1^{\tilde{L}'} \ldots T_{i-1}^{\tilde{L}'} \rangle$. Thus we can apply Lemma 4, to be described next, which assures that the resultant sequence after swapping $T_i^{\tilde{L}'}$ to the ith location comes with a shorter or equal finish time. This explains

$$f_{L_{pre}\oplus\tilde{L}'} \le f_{L_{pre}\oplus L'} = f_{L_{pre}\oplus L}.$$

$\tilde{L}$ is a sequence with the same order of $\tilde{L}'$, but the ready times and deadlines of the tasks are recovered to their original values. Each task in $\tilde{L}$ can be started no later than the starting time of the same task in $\tilde{L}'$. Consequently,

$$f_{L_{pre} \oplus \tilde{L}} \leq f_{L_{pre} \oplus L}.$$

By Corollary 1, we have

$$f_{L_{pre} \oplus \tilde{L} \oplus L_{post}} \leq f_{L_{pre} \oplus L \oplus L_{post}}.$$

$\square$

Lemma 4 Assume that $S1 \oplus S2 \oplus \langle T_j \rangle \oplus S3$ is feasible, where $S1, S2$, and $S3$ are sequences. If $T_j \lhd S2$, then $f_{S1 \oplus \langle T_j \rangle \oplus S2 \oplus S3} \leq f_{S1 \oplus S2 \oplus \langle T_j \rangle \oplus S3}$.

PROOF: We will prove the theorem by induction on $|S2|$. When $|S2| = 0$, it is vacuously true. Assume that it is true when $|S2| = k$. We would like to show that it is true when $|S2| = k + 1$. Let $S2 = \langle T_i \rangle \oplus S2'$, where $|S2'| = k$; i.e.,

$$S1 \oplus S2 \oplus \langle T_j \rangle \oplus S3 = S1 \oplus \langle T_i \rangle \oplus S2' \oplus \langle T_j \rangle \oplus S3.$$

We can view $S1 \oplus \langle T_i \rangle$ as a single sequence, and because $|S2'| = k$, by induction hypothesis, we have

$$f_{S1 \oplus \langle T_i \rangle \oplus \langle T_j \rangle \oplus S2' \oplus S3} \leq f_{S1 \oplus \langle T_i \rangle \oplus S2' \oplus \langle T_j \rangle \oplus S3}.$$

By definition,

$$
\begin{aligned}
f_{S1 \oplus \langle T_i \rangle \oplus \langle T_j \rangle} &= max(max(f_S, r_i) + c_i, r_j) + c_j \\
&= max(f_S + c_i + c_j, r_i + c_i + c_j, r_j + c_j)
\end{aligned}
$$

Since $T_j \lhd S2$, which indicates that $T_j \lhd T_i$, we have $r_j \leq r_i$, and $r_j + c_j \leq r_i + c_i + c_j$.

$$f_{S1 \oplus \langle T_i \rangle \oplus \langle T_j \rangle} = max(f_S + c_i + c_j, r_i + c_i + c_j)$$

231

On the other hand,

$$f_{S_1 \oplus \langle T_j \rangle \oplus \langle T_i \rangle} = max(max(f_S, r_j) + c_j, r_i) + c_i$$
$$= max(f_S + c_i + c_j, r_j + c_i + c_j, r_i + c_i)$$

Because $r_j + c_i + c_j \le r_i + c_i + c_j$, we have

$$f_{S_1 \oplus \langle T_j \rangle \oplus \langle T_i \rangle} \le f_{S_1 \oplus \langle T_i \rangle \oplus \langle T_j \rangle}.$$

By Corollary 1,

$$f_{S_1 \oplus \langle T_j \rangle \oplus \langle T_i \rangle \oplus S_2' \oplus S_3} \le f_{S_1 \oplus \langle T_i \rangle \oplus \langle T_j \rangle \oplus S_2' \oplus S_3}.$$

Therefore,

$$f_{S_1 \oplus \langle T_j \rangle \oplus S_2 \oplus S_3} \le f_{S_1 \oplus S_2 \oplus \langle T_j \rangle \oplus S_3}.$$

$\square$

3.3   Dominance Theorem

The super sequence is constructed for the feasibility test of a task set. If a task set is feasible, we say that there exists a *full schedule* of the task set. There may exist more than one full schedule for a given task set. An *optimal full schedule* is a full schedule whose finish time is shortest among all the possible full schedules. Note that a full schedule is a feasible instance. In this section, we prove that if a task set is feasible, there exists an optimal full schedule conforming to the super sequence.[*] Hence, the super sequence provides a valid and pruned search space for deriving the optimal full schedule of a task set.

---

[*]In [2], Erschler *et al.'s* theorem implied a similar result: if a task set is feasible, there exists a full schedule in the dominant set. Our theorem further shows that there exists such a full schedule, with the minimum finish time among all full schedules, that conforms to the super sequence. We prove the existence of such an optimal full schedule in a more systematic way.

Theorem 3 Assume that the task set Γ is feasible and ρ is an optimal full schedule of Γ. Let T_α and T_β be two top tasks of ρ such that $T_\alpha \prec T_\beta$. If $\langle T_\beta, T_\alpha \rangle$ conforms to ρ, then there exists another optimal full schedule ρ' such that $\langle T_\alpha, T_\beta \rangle$ conforms to ρ'.

PROOF: T_α and T_β are two top tasks. Let F be a frame such that $b_F = s_\beta$ and $e_F = f_\alpha$. $T_\alpha \prec T_\beta$ means $b_F = s_\beta \geq r_\beta \geq r_\alpha$, and $e_F = f_\alpha \leq d_\alpha$. If there exists a task T_x such that $F \sqcup T_x$, then $T_\alpha \sqcup T_x$ too. This contradicts to the fact that T_α is a top task. Hence F cannot contain any task. By the Leading Theorem, there exists another sequence ρ' such that $\langle T_\alpha, T_\beta \rangle$ conforms to ρ', and both $|\rho'| = |\rho|$ and $f_{\rho'} \leq f_\rho$ hold, which means ρ' is an optimal full schedule too. $\qquad\square$

When two tasks match each other, it dose not matter which task is executed first. This gives rise to the following Corollary.

Corollary 2 Assume that the task set Γ is feasible and ρ is an optimal full schedule of Γ. Also assume that $\langle h_1, \ldots, T_\beta, T_\alpha, \ldots, h_t \rangle$, the subsequence of the top tasks in ρ, conforms to ρ. If $T_\alpha \lhd T_\beta$, then there exists another optimal full schedule ρ' such that $\langle h_1, \ldots, T_\alpha, T_\beta, \ldots, h_t \rangle$ conforms to ρ'.

PROOF: Theorem 3 holds when $T_\alpha \lhd T_\beta$, because when two tasks match each other, the execution order of the two tasks is arbitrary. Also by looking at the adjustment process of Leading Theorem, we can find that the tasks located before and after T_α and T_β have not been adjusted. This verifies the corollary. $\qquad\square$

Corollary 3 Let $H = \langle h_1, h_2, \ldots, h_t \rangle$ be top tasks of the task set Γ such that $h_1 \lhd h_2 \lhd \ldots \lhd h_t$. If Γ is feasible, there exists an optimal full schedule ρ' to which H conforms.

PROOF: Since Γ is feasible, there exists an optimal full schedule ρ. Let $K = \langle k_1, k_2, \ldots, k_t \rangle$ be a sequence which is a permutation of H such that K conforms to ρ. We would like to adjust the order of the tasks in K so that K is transformed successively into H. We locate the corresponding task of h_x in K, where x is chosen in the order of 1 through t, and adjust

233

it to the xth position in K by consecutively swapping h_x with its left neighbor. This leads to the sequence H. During the swapping, h_x always weakly leads to its left neighbor, for $h_1, \ldots, h_{x-1}$ are in positions $1, \ldots, x-1$. By Corollary 2, there always exists an optimal full schedule to which the intermediate resultant sequences conform. Therefore, there exists an optimal full schedule ρ' to which $\langle h_1, h_2, \ldots, h_t \rangle$ conforms. $\square$

Given an optimal full schedule ρ, we can always obtain another optimal full schedule ρ' in which the top tasks are ordered according to the weakly leading relations by Corollary 3. Therefore the rule R1 is verified.

Before we can go further, the following definitions are useful. Let h_k be a top task and T_x a nontop task of a sequence S. We say that (h_k, T_x) is a *disorder pair* of S if $\langle h_k, T_x \rangle$ conforms to S and $T_x \prec h_k$. Similarly, (T_x, h_k) is a disorder pair of S if $\langle T_x, h_k \rangle$ conforms to S and $h_k \prec T_x$. The *disorder degree* of S is defined to be the number of disorder pairs in S.

Theorem 4 Assume that the task set Γ is feasible and ρ is an optimal full schedule of Γ. Let $h_1 \lhd h_2 \lhd \ldots \lhd h_t$ be top tasks and T_x a nontop task of ρ. Assume that $\rho = L_{pre} \oplus L \oplus L_{post}$ such that

$$\langle h_1, \ldots, h_{k-1} \rangle \; conforms \; to \; L_{pre},$$
$$\langle h_{k+1}, \ldots, h_t \rangle \; conforms \; to \; L_{post}.$$

We have the following properties:
(1) if $T_x \prec h_k$ and $L = \langle h_k, \ldots, T_x \rangle$, then there exists another optimal full schedule $\rho' = L_{pre} \oplus \bar{L} \oplus L_{post}$ such that $\bar{L}$ is a permutation of L, and $\langle T_x, h_k \rangle$ conforms to $\bar{L}$; besides, the disorder degree of ρ' is less than that of ρ
(2) if $h_k \prec T_x$ and $L = \langle T_x, \ldots, h_k \rangle$, then there exists another optimal full schedule $\rho' = L_{pre} \oplus \bar{L} \oplus L_{post}$ such that $\bar{L}$ is a permutation of L, and $\langle h_k, T_x \rangle$ conforms to $\bar{L}$; besides, the disorder degree of ρ' is less than that of ρ

PROOF: We will prove (1) first. Let F be a frame with $b_F = s_{h_k}$ and $e_F = f_x$. Since $T_x \prec h_k$, $e_F = f_x \leq d_x \leq d_{h_k}$. Also $b_F = s_{h_k} \geq r_{h_k}$. If there exists a task T_{w_i} located between h_k and

234

T_x such that $F \sqcup T_{w_i}$, then $h_k \sqcup T_{w_i}$ too. This contradicts to the fact that h_k is a top task. The condition of the Leading Theorem is satisfied. Hence there exists a sequence $\tilde{L}$ which is a permutation of L such that $\langle T_x, h_k \rangle$ conforms to $\tilde{L}$ and $f_{L_{pre} \oplus \tilde{L} \oplus L_{post}} \leq f_\rho$. Therefore, $L_{pre} \oplus \tilde{L} \oplus L_{post}$ is also an optimal full schedule. Now let us look at Figure 4(iv). This is the schedule after the adjustment process of the Leading Theorem is made. For the tasks whose deadlines are less than e_F, they all lead to h_k. Note that the disorder is a relationship defined between a nontop task and a top task, and h_k is the only top task in the frame F. Therefore, no new disorder pairs with h_k are introduced among these tasks. Similarly, for the tasks whose ready times are greater than b_F, they are all led by h_k. Therefore, no new disorder pairs are introduced. As for the tasks otherwise, including T_x and h_k, whose deadlines are greater than or equal to e_F and ready times less than or equal to b_F, they can be ordered arbitrarily. Hence, we can position T_x before h_k, and remove the disorder pairs, if any, in these tasks by rearranging the proper orders for them. Thus the disorder degree of $\tilde{L}$ is decremented by at least one. So the disorder degree of ρ' is less than that of ρ. Property (2) holds for the same reason. □

Note that T_x does not match h_k or h_{k+1}; otherwise T_x is also a top task, which contradicts our assumption.

Theorem 5 Assume that the task set Γ is feasible and ρ is an optimal full schedule of Γ. Let $h_1 \lhd h_2 \lhd \dots \lhd h_t$ be top tasks of ρ. There exists an optimal full schedule ρ' such that $\langle h_1, h_2, \dots, h_t \rangle$ conforms to ρ', and for any nontop task T_x such that $\langle h_k, T_x, h_{k+1} \rangle$ conforms to ρ', either $T_x \sqcup h_k$ or $T_x \sqcup h_{k+1}$.

PROOF: Assume that T_x is a nontop task such that $\langle h_k, T_x, h_{k+1} \rangle$ conforms to ρ'. If T_x does not contain h_k and T_x does not contain h_{k+1}, then either $T_x \prec h_k$ or $h_{k+1} \prec T_x$. Hence, either (h_k, T_x) or (T_x, h_{k+1}) is a disorder pair. We can eliminate it through Theorem 4, and the disorder degree is decremented by at least one. Whenever there is a disorder pair in the schedule, we can always apply Theorem 4 to eliminate it. The disorder degree is decremented in this way until finally reaching zero. Hence, $\langle h_k, T_x, h_{k+1} \rangle$ conforming to ρ' implies that T_x is not leading to h_k and h_{k+1} is not leading to T_x. The only possibilities are either $T_x \sqcup h_k$

or $T_x \sqcup h_{k+1}$. $\qquad$ □

Theorem 5 confirms the validity of rules R1 and R2.

Theorem 6 (Dominance Theorem) If a task set Γ is feasible, there exists an optimal full schedule ρ such that ρ conforms to the super sequence of Γ.

PROOF: In Theorem 5, we verify the existence of the optimal full schedule such that the top tasks are ordered according to their weakly leading relations, and the nontop tasks are located in the appropriate subsets between top tasks. The only work left is to order the nontop tasks in each subset. The adjustment process of the Leading Theorem can be applied, and the resultant order is exactly specified by rule R3. So we can conclude that there exists an optimal full schedule ρ which conforms to the super sequence. $\qquad$ □

3.4 Conformation Theorem

If there is no task rejected in ρ, there exists an optimal full schedule conforming to the super sequence of Γ. However, if Γ is not feasible, some tasks in Γ should be rejected. The dominant rules are developed based on the assumption that no task is rejected. When tasks are allowed to be rejected, the situation is different. The issue to be raised is whether the decent solution for feasibility test can be applied to our optimization problem. Remember that by optimization we mean that the number of rejected tasks in the schedule is minimized and then the finish time of the schedule is also minimized. When a task set is feasible, the optimal schedule is also the optimal full schedule. The difficulties are addressed in the next section, followed by the approach and proof to solving the difficulties.

3.4.1. Difficulties

We wish to make use of the super sequence as search space in our scheduling problem. The difficulties are twofold.

First, when a task is allowed to be rejected, the dominant rules specifying the relations among containing tasks and contained tasks need to be modified, because the rules are

236
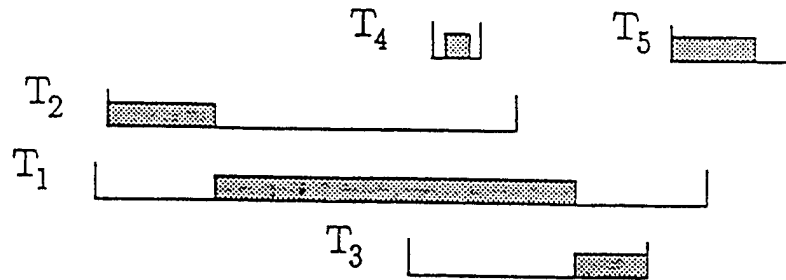
Figure 5: The optimal schedule may not conform to the super sequence.

developed based on the assumption that no task is to be rejected. The new rules can become quite complicated. Let's look at the example depicted in Figure 5. Assume that the task set is

$$\Gamma = \{T_1, T_2, T_3, T_4, T_5\},$$

and the super sequence of the task set is

$$\Delta = \langle T_1, T_2, T_3, \mathbf{T_4}, T_2, T_3, T_1, \mathbf{T_5} \rangle.$$

The top tasks are typed in bold letters for emphasis. Γ is not feasible. We can see that one possibility of the optimal schedule could be

$$\rho_0 = \langle T_2, T_1, T_3, T_5 \rangle.$$

Apparently, ρ_0 does not conform to Δ. One may be able to show that another optimal schedule $\langle T_2, T_4, T_3, T_5 \rangle$ conforms to Δ. However, given an arbitrary task set, it is not guaranteed that one is always able to do so. In the example, T_4 is rejected. If we recompute the super sequence without T_4, we would get a different super sequence. The new super sequence would be

$$\Delta_0 = \langle T_1, \mathbf{T_2}, T_1, \mathbf{T_3}, T_1, \mathbf{T_5} \rangle,$$

to which ρ_0 conforms. This gives a great difficulty. It seems that we need to check against each task. Construct a super sequence in condition that the task is accepted, and

construct another super sequence in condition that the task is rejected. In general, we need to construct 2^n super sequences in this way. This is too formidable to schedule, considering that the number of instances in each individual super sequence can be exponential to the size of the task set.

Secondly, while rejecting a nontop tasks does not affect much, rejecting a top task could affect the duplication and positions of the nontop tasks or might even result in some new top tasks. Thus, the super sequences can be totally different. Look at the same example in Figure 5. The rejection of T_4 results in two more top tasks, i.e., T_2 and T_3. This makes Δ_0 completely different from Δ.

We propose a *swapping and replacing* procedure to overcome the difficulties. The procedure would be described and verified in the following section.

3.4.2 Approach and Proof

The idea of our approach is stated briefly below, followed a formal proof. Let Γ and Δ be the original task set, and the super sequence of the task set respectively. It is clear that there exists an optimal schedule, which is unknown to us, for any task set. Let Γ_0 be the task set which is composed of the tasks of the unknown optimal schedule, and Δ_0 be the super sequence of Γ_0. Γ_0 and Δ_0 are also unknown to us. As mentioned above, Δ_0 might be quite different from Δ. Notice that the unknown optimal schedule of Γ is also an optimal (full) schedule of Γ_0. Since Γ_0 is feasible, by the Dominance Theorem, there exists an optimal full schedule for Γ_0, say ρ_0, such that ρ_0 conforms to Δ_0. Our problem is that we are not able to compute ρ_0 from Δ_0, because Δ_0 is unknown. We are able to compute Δ from Γ by applying the dominant rules. The swapping and replacing procedure exploits the way to adjust the order of tasks in ρ_0 and to replace some tasks if necessary, so as to transform ρ_0 into a new schedule ρ such that ρ is also an optimal schedule and best of all ρ conforms to an instance of Δ. For the sake of simplicity, we will say a schedule conforms to Δ, when the schedule conforms to an instance of Δ. So we can use Δ as a valid search space when scheduling Γ.

In the example of Figure 5, we transform ρ_0 into

$$\rho = \langle T_2, T_4, T_3, T_5 \rangle.$$

238

This example is so simplified that the existence of ρ can be verified by mere intuition. However, the reasoning is far more complicated than it appears at the first glance. We are going to prove in the following theorem that such an optimal schedule ρ that conforms to Δ always exists. The corresponding lemmas are presented in the next section.

Theorem 7 (Conformation Theorem) Given a task set $\Gamma = \{T_1, T_2, ..., T_n\}$, there exists an optimal schedule ρ such that ρ conforms to the super sequence Δ of Γ.

PROOF: Given any task set Γ, there exists at least one optimal schedule, which is unknown to us. Assume that we need to reject w tasks from Γ to make a feasible schedule. Let Γ_0 be the task set which is composed of the tasks in the unknown optimal schedule. Γ_0 is a subset of Γ. The super sequence of Γ_0 is denoted by Δ_0. In addition, we use Γ_j, $0 \le j \le w$, to represent a task set derived by adding j tasks into Γ_0, Δ_j the super sequence of Γ_j, and ρ_j an optimal schedule of Γ_j. When we say adding j tasks into Γ_0, we mean that the resultant task set Γ_j is composed of distinct tasks and Γ_j is a subset of Γ. In particular, Γ_w is Γ. We will prove by induction on w to show that there exists an optimal schedule ρ_w for Γ conforming to Δ_w.

Base step $w = 0$: there is no task rejected. $\Gamma = \Gamma_0$. Since Γ is feasible, by the Dominance Theorem, there exists an optimal (full) schedule ρ_0 for Γ such that ρ_0 conforms to Δ_0.

Induction hypothesis: assume that the theorem holds when $w = j$, i.e., $|\rho_0| = n - j$. For the task set Γ_j which is derived by adding j tasks into Γ_0, there exists an optimal schedule ρ_j for Γ such that ρ_j conforms to Δ_j. Notice that $|\rho_j| = |\rho_0|$, and $|\Gamma_j| = |\Gamma_0| + j$.

Now consider the case when $w = j + 1$, i.e., $|\rho_0| = n - (j + 1)$. We need to reject $j + 1$ tasks to make a feasible schedule. There exists an optimal schedule ρ_j for Γ conforming to Δ_j by induction hypothesis. We want to show that, by swapping and replacing the tasks in ρ_j, the resultant sequence ρ_{j+1} conforms to Δ_{j+1}; besides, $|\rho_{j+1}| = |\rho_j|$, and $f_{\rho_{j+1}} \le f_{\rho_j}$, which implies that ρ_{j+1} is also an optimal schedule for Γ. Let T_x be the task added into Γ_j to make Γ_{j+1}. So, $\Gamma_j \cup \{T_x\} = \Gamma_{j+1}$. There are two possibilities when adding T_x.

If T_x is a nontop task of Γ_{j+1}, adding T_x does not add a top task into Γ_j. The orders of the top tasks in both Δ_{j+1} and Δ_j derived through rule R1 are exactly the same. Rule R2 specifies the relation between a nontop task and a top task. Adding a nontop task T_x

239

does not affect the relations between the already existent nontop tasks and top tasks. The positions (duplicates) of the already existent nontop tasks in Δ_j are preserved in Δ_{j+1}. Rule R3 specifies how to arrange the order of the nontop tasks within each subset. Again adding a nontop task T_x does not alter the orders of the already existent nontop tasks in each subset in Δ_j. Therefore, if the task being added is a nontop task, Δ_j is a subsequence of Δ_{j+1}. Let us look at the example in Figure 5. Assume that Γ_j and Γ_{j+1} are

$$\Gamma_j = \{T_2, T_3, T_4, T_5\}, \text{ and}$$

$$\Gamma_{j+1} = \{T_1, T_2, T_3, T_4, T_5\},$$

where T_1 is a nontop task. The corresponding super sequences would be

$$\Delta_j = \langle T_2, T_3, T_4, T_2, T_3, T_5 \rangle, \text{ and}$$

$$\Delta_{j+1} = \langle T_1, T_2, T_3, T_4, T_2, T_3, T_1, T_5 \rangle.$$

We can see in the example how Δ_j conforms to Δ_{j+1}.

Otherwise, T_x is a top task of Γ_{j+1}. T_x does not contain other tasks in Γ_{j+1}. Two situations are possible.

(i) T_x is not contained by other tasks. The number of top tasks in Γ_{j+1} is one more than that of the top tasks in Γ_j. The order of the top tasks in Δ_j is preserved in Δ_{j+1}, since the relations of the top tasks are not altered by adding T_x. Furthermore, T_x does not alter any existent orders among the nontop tasks and top tasks, or among the orders between the nontop tasks and nontop tasks, specified by rules R2 and R3, respectively. Therefore, Δ_j is a subsequence of Δ_{j+1}. Let us look at the example in Figure 6. Assume that Γ_j and Γ_{j+1} are

$$\Gamma_j = \{T_1, T_2, T_4, T_5\}, \text{ and}$$

$$\Gamma_{j+1} = \{T_1, T_2, T_3, T_4, T_5\},$$

where T_3 is a top task not contained by other tasks. The corresponding super sequences would be

$$\Delta_j = \langle T_1, T_2, T_1, T_4, T_5, T_4 \rangle, \text{ and}$$
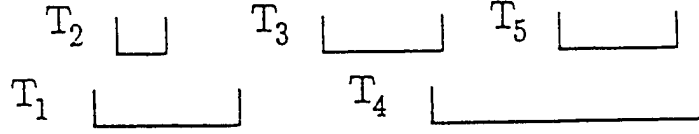
Figure 6: The added top task T_3 is not contained by other tasks.

$$\triangle_{j+1} = \langle T_1, T_2, T_1, T_3, T_4, T_5, T_4 \rangle.$$

We can see in the example how $\triangle_j$ conforms to $\triangle_{j+1}$.

(ii) T_x is contained by some top tasks and/or nontop tasks of Γ_j. Let the top tasks of Γ_j containing T_x be $g_1, \ldots, g_m$, indexed in the weakly leading order. This situation is more complicated, because $g_i, i = 1, \ldots, m$, turn out to be nontop tasks in Γ_{j+1}. There exists a total ordering of them by weakly leading relations, because there is no containing relations among g_i. By rule R1, the super sequence of Γ_{j+1} can be expressed as

$$\triangle_{j+1} = \langle \ldots, h_{k-1}, \ldots, g_1, \ldots, g_m, \ldots, h_k, \ldots, g_1, \ldots, g_m, \ldots, h_{k+1}, \ldots \rangle,$$

where $h_1, \ldots, h_{k-1}, h_k, h_{k+1}, \ldots$ are the top tasks in Γ_{j+1}, and in particular, h_k represents T_x. By rule R3, the super sequence of Γ_{j+1} can also be expressed as

$$\triangle_{j+1} = \langle \ldots, h_{k-1}, B_{k-1,\overline{k}}, \underbrace{B_{k-1,k}. B_{\overline{k-1},k}, h_k, B_{k,\overline{k+1}}, B_{k,k+1}}_{\Omega'_{j+1}}, B_{\overline{k},k+1}, h_{k+1}, \ldots \rangle, \qquad (6)$$

$$\underbrace{\phantom{B_{k-1,\overline{k}}, B_{k-1,k}. B_{\overline{k-1},k}, h_k, B_{k,\overline{k+1}}, B_{k,k+1}, B_{\overline{k},k+1}}}_{\Omega_{j+1}}$$

where Ω_{j+1} represents the subsequence of $\triangle_{j+1}$ between h_{k-1} and h_{k+1}, excluding h_{k-1} and h_{k+1}, as depicted above, and $\Omega_{j+1} = B_{k-1,\overline{k}} \oplus \Omega'_{j+1} \oplus B_{\overline{k},k+1}$, where $\oplus$ means concatenation of sequences. Remember that $g_1, \ldots, g_m$ are top tasks of Γ_j. All the top tasks in Γ_j are in the order of $h_1, \ldots, h_{k-1}, g_1, \ldots, g_m, h_{k+1}, \ldots$ by the weakly leading relations. By rule R1, the super sequence of Γ_j can be expressed as

$$\triangle_j = \langle \ldots, h_{k-1}, \underbrace{\ldots, g_1, \ldots, g_m, \ldots}_{\Omega_j}, h_{k+1}, \ldots \rangle, \qquad (7)$$

where Ω_j represents the subsequence of Δ_j between h_{k-1} and h_{k+1}, excluding h_{k-1} and h_{k+1}, as depicted above. Notice that in Equations 6 and 7, the subsequences before h_{k-1} of both Δ_{j+1} and Δ_j are exactly the same, because the addition of the top task h_k, or T_x, affects only the subsequence between h_{k-1} and h_{k+1}. Similarly, the subsequences after h_{k+1} of both Δ_{j+1} and Δ_j are exactly the same too. Hence an instance of Δ_{j+1} will differ from an instance of Δ_j only in the the subsequences of Ω_{j+1} and Ω_j.

Now we would like to check what tasks in Ω_j should follow immediately after h_{k-1}. By Lemma 7, all the tasks in Ω_j can be found in $\Omega_{j+1} - h_k$. So we only need to check the tasks in $\Omega_{j+1} - h_k$. If a task contains h_{k-1} but not h_k, the task must not contain g_1. Because g_1 contains h_k, any task which contains g_1 should also contain h_k. When constructing the subsequence of Δ_j between h_{k-1} and g_1, by rule R2, all the tasks which appear in $B_{k-1,\bar{k}}$ of Δ_{j+1} should follow immediately after h_{k-1}; and by rule R3, the order of the subsequence is exactly the same as $B_{k-1,\bar{k}}$.

One may observe that some tasks in $B_{k-1,k}$ contain h_{k-1} but do not contain g_1, so they would also be positioned between h_{k-1} and g_1. These tasks would follow after the tasks of $B_{k-1,\bar{k}}$. This is because they do contain h_k and hence have greater deadlines than those tasks in $B_{k-1,\bar{k}}$. For the same reason, all the tasks which appear in $B_{\bar{k},k+1}$ of Δ_{j+1} should be located immediately before h_{k+1} when constructing Δ_j, and the order is the same. Hence, the Δ_j can be further expressed as

$$\Delta_j = \langle \ldots, h_{k-1}, B_{k-1,\bar{k}}, \underbrace{\ldots, g_1, \ldots, g_m, \ldots}_{\Omega_j'}, B_{\bar{k},k+1}, h_{k+1}, \ldots \rangle, \tag{8}$$

where Ω_j' represents the subsequence between $B_{k-1,\bar{k}}$ and $B_{\bar{k},k+1}$ of Δ_j, excluding $B_{k-1,\bar{k}}$ and $B_{\bar{k},k+1}$. We have $\Omega_j = B_{k-1,\bar{k}} \oplus \Omega_j' \oplus B_{\bar{k},k+1}$. By Lemma 9, all the tasks in Ω_j' are either in $B_{k-1,k}$ or in $B_{\overline{k-1},k}$.

Let us look at an example in Figure 7. The task set in the figure is Γ_{j+1}. And $\Gamma_{j+1} - h_k$ would be Γ_j. g_1 and g_2 contain only h_k. So g_1 and g_2 are classified as nontop tasks in Γ_{j+1},

242

h_{k-1} ⎵ h_k ⎵ h_{k+1} ⎵⎵

g_1 ⎵⎵⎵ T_5 ⎵⎵

g_2 ⎵⎵

T_4 ⎵⎵⎵

T_2 ⎵⎵⎵⎵⎵

T_3 ⎵⎵⎵⎵⎵

T_1 ⎵⎵⎵⎵⎵⎵
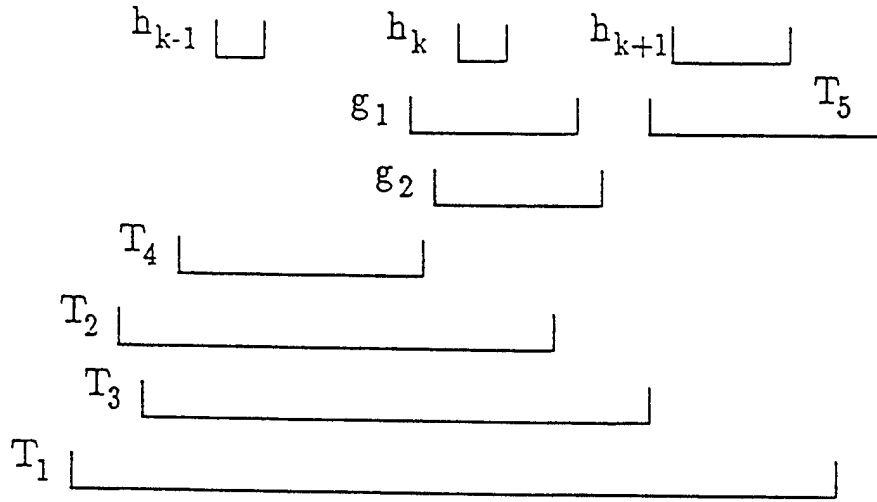
Figure 7: The added top task h_k is contained by other tasks.

and as top tasks in Γ_j. We can compute the super sequences as follows.

$$\Delta_{j+1} = \langle T_1, T_2, T_3, T_4, h_{k-1}, \underbrace{T_4}_{B_{k-1,\bar{k}}}, \underbrace{T_2, T_3, T_1}_{B_{k-1,k}}, \underbrace{g_1, g_2}_{B_{\overline{k-1},k}}, h_k, \underbrace{T_2, g_1, g_2, T_3}_{B_{k,\overline{k+1}}}, \underbrace{T_1}_{B_{k,k+1}}, \underbrace{T_5}_{B_{\bar{k},k+1}}, h_{k+1}, T_1, T_5 \rangle,$$

$$\underbrace{}_{\Omega'_{j+1}}$$

$$\underbrace{}_{\Omega_{j+1}}$$

$$\Delta_j = \langle T_1, T_2, T_3, T_4, h_{k-1}, \underbrace{T_4}_{B_{k-1,\bar{k}}}, \underbrace{T_2, T_3, T_1, g_1, T_3, T_1, g_2, T_3, T_1,}_{\Omega'_j}, \underbrace{T_5}_{B_{\bar{k},k+1}}, h_{k+1}, T_1, T_5 \rangle.$$

$$\underbrace{}_{\Omega_j}$$

Now going back to examine Equations 6 and 8, we can find that Δ_j and Δ_{j+1} only differ in the middle subsequences represented by Ω'_j and Ω'_{j+1}. This can also be seen in the example in Figure 7. The instances extracted from Δ_j would conform to Δ_{j+1} except the corresponding middle subsequence mentioned above. Remember that ρ_j conforms to Δ_j. We would try to adjust the order of the tasks of the subsequence in ρ_j which correspond to Ω'_j for the purpose that the resultant schedule ρ_{j+1} conforms to Δ_{j+1} and ρ_{j+1} is also an optimal schedule of Γ. The adjustment procedure, called the *swapping and replacing* method, applied to ρ_j is described below:

C1: for all tasks $T_y \in \Omega_j'$ such that $f_y \le d_{h_k}$, they are sorted by their ready times r_y.

C2: for all tasks $T_y \in \Omega_j'$ such that $s_y \ge r_{h_k}$, they are sorted by their deadlines d_y.

C3: a task can be sorted by C1 or C2 described above if the task satisfies both conditions.

C4: if there exists a task $T_y \in \Omega_j'$ such that $s_y < r_{h_k}$ and $f_y > d_{h_k}$, T_y is replaced by h_k.

We would like to show that the adjustment does make ρ_{j+1} conform to Δ_{j+1}. Remember that Δ_j and Δ_{j+1} only differ in the middle subsequences represented by Ω_j' and Ω_{j+1}'. We only swap and replace the tasks of ρ_j located in Ω_j' to derive ρ_{j+1}. Since ρ_j conforms to Δ_j, the head and the tail of ρ_{j+1} also conforms to Δ_{j+1}. So we only need to check the middle subsequence of ρ_{j+1} to see if the whole sequence of ρ_{j+1} conforms to Δ_{j+1}. The tasks adjusted by the swapping and replacing procedure are either in $B_{k-1,k}$ or in $B_{\overline{k-1},k}$ by Lemma 9. Let us first check the adjustment of C1. In Δ_{j+1}, the order of the tasks in $B_{k-1,k}$ can be determined arbitrarily according to rule R3, so it does not matter which task is located before which. And in Δ_{j+1}, the order of the tasks in $B_{\overline{k-1},k}$ is determined by their ready times. During the adjustment of C1, all the qualifying tasks are ordered according to their ready times. We know that the ready times of the tasks in $B_{k-1,k}$ are less than the ready times of the tasks in $B_{\overline{k-1},k}$. So, in the resultant schedule ρ_{j+1}, the tasks of $B_{k-1,k}$ are positioned before the tasks of $B_{\overline{k-1},k}$. This indicates that the order of these tasks in ρ_{j+1} after the adjustment of C1 conforms to Δ_{j+1}. For the same reason, the adjustment of C2 makes the order of the swapped tasks conform to Δ_{j+1}. In condition C4, if such a T_y exists, replacing T_y by h_k also conforms to Δ_{j+1}, which can be seen in Equation 6. Each task $T_y \in \Omega_j'$ satisfies one of the conditions by Lemma 11. Hence, all the tasks in the middle subsequence of ρ_{j+1} are adjusted in such a way that the order conforms to Δ_{j+1}. So ρ_{j+1} conforms to Δ_{j+1}.

Now we would like to show that ρ_{j+1}, in addition to conforming to Δ_{j+1}, can also be finished no later than ρ_j. We can view the tasks satisfying condition C1 as having the same virtual deadlines of d_{h_k}, because they all finish before this time instant. Hence, there is a total ordering among the tasks with virtual deadlines by weakly leading relations, which is achieved by sorting their ready times. By Lemma 10, the finish time of the resultant

244

schedule after the adjustment of C1 would not be greater than that of the original schedule. Similarly, the tasks satisfying condition C2 can be viewed as having the same virtual ready times of r_{h_k}, because they all start after this time instant. For the same reason, the finish time of the resultant schedule after the adjustment of C2 would not be greater than that of the original schedule. In condition C3, the qualifying tasks can be sorted in either way and does not affect the result. In condition C4, if there exists a task T_y whose computation time covers the whole window of the rejected task h_k, we may as well replace T_y by h_k, and the finish time of the resultant schedule after the adjustment of C4 would not be greater than that of the original schedule. Each task $T_y \in \Omega'_j$ satisfies one of the conditions by Lemma 11. Hence, all the tasks in the middle subsequence of ρ_{j+1} are adjusted in such a way that ρ_{j+1} would be finished no later than ρ_j. Therefore, $|\rho_{j+1}| = |\rho_j|$, and $f_{\rho_{j+1}} \leq f_{\rho_j}$. Since ρ_j is an optimal schedule of Γ, ρ_{j+1} is also an optimal schedule of Γ.

How the adjustment procedure makes the finish time shorter is illustrated by Figures 8. In Figure 8(i), both T_1 and T_3 satisfy condition C3, and T_2 satisfies condition C1. The procedure of C1 is applied to all these three tasks and makes the finish time shorter. The dotted task window frame in the figure indicates that h_k is a rejected task. In Figure 8(ii), C1 is applied to the qualifying tasks T_1 and T_2. And by C4, T_3 is replaced by h_k. This makes the finish time shorter. While it is h_k that is rejected before the adjustment, it turns out that T_3, whose computation time covers the whole window of h_k, is rejected after the adjustment.

So far, we have shown that ρ_{j+1} conforms to Δ_{j+1}, and that ρ_{j+1} is also an optimal schedule of Γ. The theorem is thus verified by the induction. It deserves our attention that we do not really apply the swapping and replacing procedure to any schedule. We just want to show the existence of the optimal schedule which is ρ_{j+1} in the context. To make it clear, the structure of the theorem is illustrated in Figure 9. □

3.4.3 Corresponding Lemmas

The lemmas used by the conformation theorem are demonstrated as follows.

Lemma 5 $B_{k-1,\bar{k}} \cap B_{k-1,k} = B_{k-1,k} \cap B_{\overline{k-1},k} = B_{k-1,\bar{k}} \cap B_{\overline{k-1},k} = \emptyset$
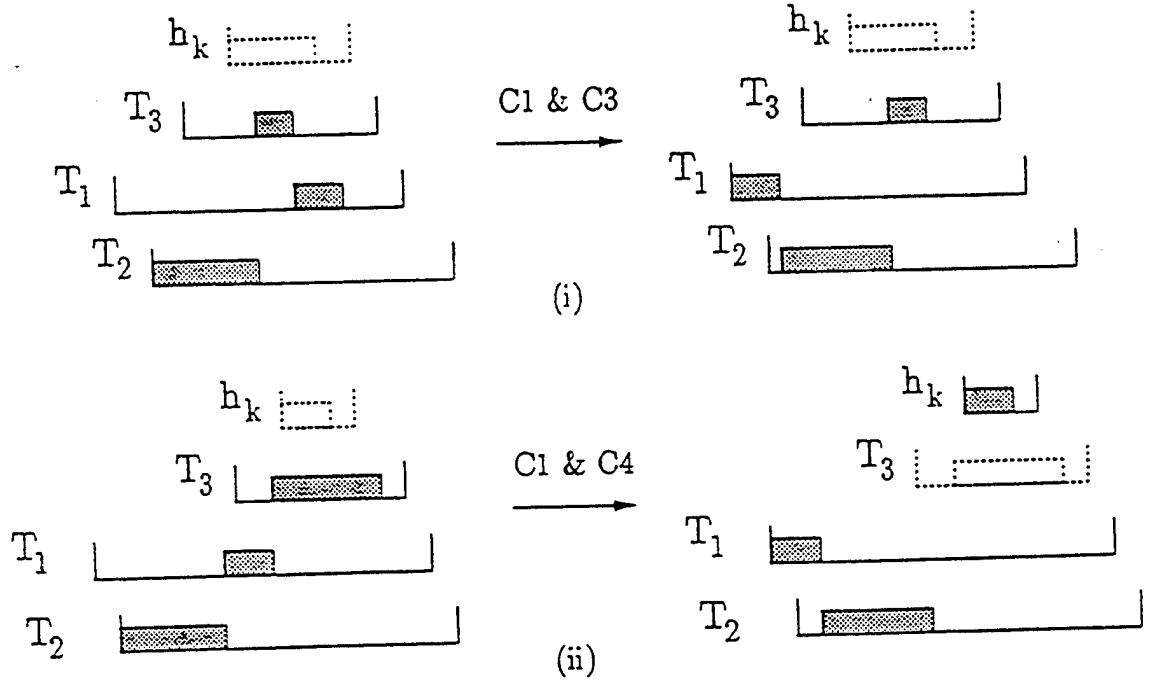
Figure 8: The swapping and replacing procedure C1, C3 and C4.

$$B_{k,\overline{k+1}} \cap B_{k,k+1} = B_{k,k+1} \cap B_{\overline{k},k+1} = B_{k,\overline{k+1}} \cap B_{\overline{k},k+1} = \emptyset.$$

PROOF: We first show that $B_{k-1,\overline{k}} \cap B_{k-1,k} = \emptyset$. Given any task $T_y \in B_{k-1,\overline{k}}$, T_y does not contain h_k by definition. Hence, T_y does not belong to $B_{k-1,k}$. So $B_{k-1,\overline{k}} \cap B_{k-1,k} = \emptyset$. The others can be proved similarly. □

Lemma 6 $B_{k-1,k} \cup B_{\overline{k-1},k} = B_{k,\overline{k+1}} \cup B_{k,k+1}$.

PROOF: We first prove that if a task $T_y \in B_{k-1,k} \cup B_{\overline{k-1},k}$, then $T_y \in B_{k,\overline{k+1}} \cup B_{k,k+1}$. Because T_y contains h_k by definition, T_y must have a location after h_k too by rule R2. $T_y \in B_{k,\overline{k+1}} \cup B_{k,k+1} \cup B_{\overline{k},k+1}$. T_y does not belong to $B_{\overline{k},k+1}$, so $T_y \in B_{k,\overline{k+1}} \cup B_{k,k+1}$. We can prove similarly that if a task $T_y \in B_{k,\overline{k+1}} \cup B_{k,k+1}$, then $T_y \in B_{k-1,k} \cup B_{\overline{k-1},k}$. So $B_{k-1,k} \cup B_{\overline{k-1},k} = B_{k,\overline{k+1}} \cup B_{k,k+1}$. □

246

Figure 9: The Structure of the Conformation Theorem

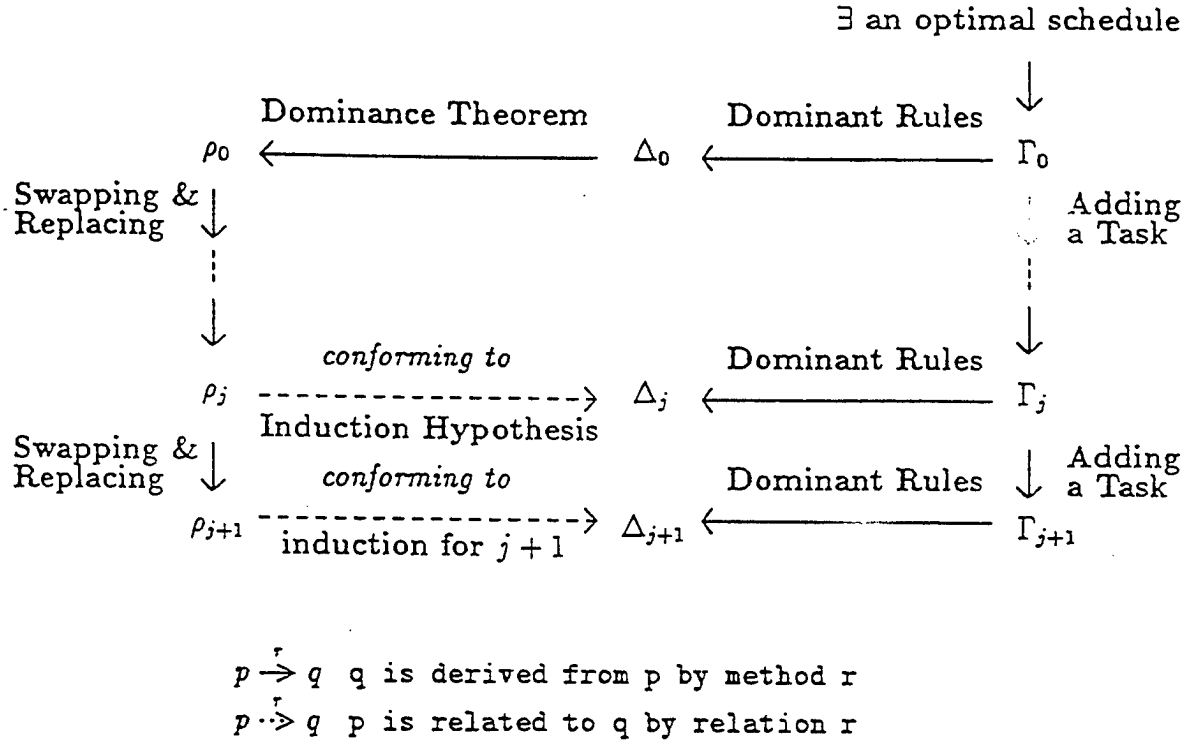Lemma 7 If and only if $T_y \in B_{k-1,\bar{k}} \cup B_{k-1,k} \cup B_{\overline{k-1},k} \cup B_{k,\overline{k+1}} \cup B_{k,k+1} \cup B_{\bar{k},k+1}$ then $T_y \in \Omega_j$.

PROOF: We will first prove the "if" part. By Lemma 6, $B_{k-1,k} \cup B_{\overline{k-1},k} = B_{k,\overline{k+1}} \cup B_{k,k+1}$. So we only need to check against $B_{k-1,\bar{k}} \cup B_{k-1,k} \cup B_{\overline{k-1},k} \cup B_{\bar{k},k+1}$. If $T_y \in B_{k-1,\bar{k}}$ or $T_y \in B_{k-1,k}$, then T_y has a location between the top tasks h_{k-1} and g_1 in Δ_j. This is because T_y contains h_{k-1}, T_y has a location after h_{k-1} by rule R2. So $T_y \in \Omega_j$. Then consider $T_y \in B_{\overline{k-1},k}$. T_y is either a top task or a nontop task in Δ_j. If T_y is a top task in Δ_j, T_y must be one of the g_i, $i = 1, \ldots, m$ by the definition of g_i. If T_y is a nontop task in Δ_j, it must contain at least one top task, which is a top task among $g_1, \ldots, g_m$, or h_{k+1} by referring to Equation 7. Notice that T_y must not contain h_{k-1} since $T_y \in B_{\overline{k-1},k}$. No matter whether T_y is a top or nontop task in Δ_j, T_y has a location in Ω_j by rule R2. If $T_y \in B_{\bar{k},k+1}$, then T_y has a location between the top tasks g_m and h_{k+1} in Δ_j by rule R2. So $T_y \in \Omega_j$.

Now we prove the "only if" part. Let T_y be a task located in Ω_j. If T_y is one of the top tasks of $g_1, \ldots, g_m$, T_y contains h_k. Either $T_y \in B_{k-1,k}$ or $T_y \in B_{\overline{k-1},k}$. Otherwise, T_y is a non-top task of Δ_j. By rule R2, T_y contains at least one of the top tasks of $h_{k-1}, g_1, \ldots, g_m, h_{k+1}$. If T_y contains one of $g_1, \ldots, g_m$, then T_y also contains h_k. So T_y contains either h_{k-1}, or h_k, or h_{k+1}. By rule R2, T_y should fall between h_{k-1} and h_k, and/or between h_k and h_{k+1}. So $T_y \in B_{k-1,\overline{k}} \cup B_{k-1,k} \cup B_{\overline{k-1},k} \cup B_{k,\overline{k+1}} \cup B_{k,k+1} \cup B_{\overline{k},k+1}$. $\quad\square$

This lemma means that the tasks in $\Omega_{j+1} - h_k$ are exactly the same tasks which are in Ω_j.

Lemma 8 If $T_y \in \Omega'_j$, T_y contains h_k.

PROOF: We would like to show that if T_y does not contain h_k, then T_y does not belong to Ω'_j. Since g_i, $i = 1, \ldots, m$, contains h_k, that T_y does not contain h_k means that T_y does not contain g_i either. Hence T_y can not have a location in the subsequence between g_1 and g_m in Ω'_j. The only possible locations of T_y to fall in Ω'_j are either between h_{k-1} and g_1, or between g_m and h_{k+1}. If T_y falls between h_{k-1} and g_1, that T_y does not contain g_1 implies that T_y contains h_{k-1} by rule R2. That is $T_y \in B_{k-1,\overline{k}}$. A nontop task cannot have duplicate positions in the same region between two adjacent top tasks. $B_{k-1,\overline{k}}$ is located between h_{k-1} and g_1 by Equation 8. T_y does not have a location in the head of Ω'_j before g_1. For the same reason, T_y does not have a location in the tail of Ω'_j after g_m. So T_y does not belong to Ω'_j. Therefore, if $T_y \in \Omega'_j$, T_y contains h_k. $\quad\square$

Lemma 9 If $T_y \in \Omega'_j$, $T_y \in B_{k-1,k} \cup B_{\overline{k-1},k} = B_{k,\overline{k+1}} \cup B_{k,k+1}$.

PROOF: If $T_y \in \Omega'_j$, then $T_y \in \Omega_j$. By Lemma 7, $T_y \in B_{k-1,\overline{k}} \cup B_{k-1,k} \cup B_{\overline{k-1},k} \cup B_{k,\overline{k+1}} \cup B_{k,k+1} \cup B_{\overline{k},k+1}$. We know that T_y contains h_k by Lemma 8, so $T_y \in B_{k-1,k} \cup B_{\overline{k-1},k} \cup B_{k,\overline{k+1}} \cup B_{k,k+1}$. Also by Lemma 6, we have $T_y \in B_{k-1,k} \cup B_{\overline{k-1},k} = B_{k,\overline{k+1}} \cup B_{k,k+1}$. $\quad\square$

Lemma 10 Assume that $S = L_{pre} \ominus L \ominus L_{post}$ is a feasible sequence, where $L = \langle T_{x_1}, T_{x_2}, \ldots, T_{x_i} \rangle$. If there exists a sequence $\bar{L} = \langle T_{y_1}, T_{y_2}, \ldots, T_{y_i} \rangle$ such that $\bar{L}$ is a permutation of L and the tasks of $\bar{L}$ are ordered by the weakly leading relation. We have $f_{L_{pre} \ominus \bar{L} \ominus L_{post}} \leq f_{L_{pre} \ominus L \ominus L_{post}}$.

PROOF: We bubble sort the tasks of L in weakly leading order. The swapping only occurs between two adjacent tasks. For each swapping, we apply the Leading Theorem to the adjacent tasks, which correspond to T_β and T_α respectively in the theorem. No other tasks lie in between the two tasks during each individual swapping. So the finish time of the resultant schedule is not greater than that of the original schedule according to the Leading Theorem.
□

Lemma 11 A task $T_y \in \Omega_j'$ should satisfy one of the conditions C1, C2, C3 or C4.

PROOF: If $T_y \in \Omega_j'$, then $T_y \in B_{k-1,k} \cup B_{\overline{k-1},k}$ by Lemma 9, which implies that $T_y \sqcup h_k$. We have $r_y < r_{h_k}$ and $d_y > d_{h_k}$. There are four possibilities.

(i) $s_y < r_{h_k}$ and $f_y > d_{h_k}$: C4 is satisfied.

(ii) $s_y \geq r_{h_k}$ and $f_y \leq d_{h_k}$: C3 is satisfied.

(iii) $s_y < r_{h_k}$ and $f_y \leq d_{h_k}$: C1 is satisfied.

(iv) $s_y \geq r_{h_k}$ and $f_y > d_{h_k}$: C2 is satisfied.

□

3.5   Set-Scheduler Algorithm

By Conformation Theorem, we have shown that there exists an optimal schedule which conforms to the super sequence Δ. Hence, we can use Sequence-Scheduler to schedule for each instance in the super sequence, and pick up the best one. Since Sequence-Scheduler obtains the optimal schedule for each instance, we end up with the optimal schedule for the task set. The algorithm for scheduling a task set is given in Figure 10. The Sequence-Scheduler takes $O(n^2)$ time for each instance, while there are

$$N = \prod_{q=1}^{q=t} (q+1)^{n_q}$$

instances to check in the super sequence as illustrated in the previous section. The time complexity of Set-Scheduler algorithm is thus $O(N * n^2)$.

Algorithm <u>Set-Scheduler</u>:

Input: a task set $\Gamma = \{T_1, T_2, ..., T_n\}$
Output: the optimal schedule ρ for Γ
compute the super sequence Δ for Γ
$\rho := \langle\rangle$
for each instance I in the super sequence Δ
    invoke <u>Sequence-Scheduler</u> to compute the optimal schedule $\sigma(n)$ of I
    if $(|\rho| < |\sigma(n)|)$ or
        $(|\rho| = |\sigma(n)|$ and $f_\rho > f_{\sigma(n)})$
        $\rho := \sigma(n)$
    endif
endfor

Figure 10: Set-Scheduler Algorithm

4 Evaluation

Experiments are conducted to compare the performance of Set-Scheduler with those of the well-known Earliest-Deadline-First and Least-Laxity-First heuristic algorithms. The relations among the tasks are important for the schedulability of the tasks. To study the differences between different cases, we allow the variation of the computation times, and the interarrival times, which are the time intervals between the ready times of two consecutive tasks. Tasks in a task set are generated in non-descending order by their ready times. The parameters of the experiments are random variables with truncated normal distribution, as shown in Figure 11. If the computation time of a task is greater than its window length, the computation time is truncated to its window length. Such a truncation is not applied to the interarrival times.

The mean of Window is fixed. Computation time ratio is the ratio of the computation time to the window length. The mean of Interarrival time ranges from 10% to 100% of the mean of Window. The standard deviation of these three random variables are set to be

parameters	mean
Window length	10.0
Computation time ratio	0.25 0.5 0.75
Interarrival time	1.0, 2.0, ..., 10.0

Figure 11: Parameters of the experiments

20%, 50%, and 80% of their means. For simplicity, the ratios of the three random variables are set to be the same for each individual experiment. For each experiment with different parameters, 100 task sets, each with 12 tasks, are generated for scheduling.

We compare the performance of these algorithms by (1) Percentage of accepted tasks: the number of accepted tasks by the algorithm over the number of the tasks of the optimal schedule by exhaustive search; (2) Success ratio: the number of times that the algorithm comes up with an optimal schedule in the 100 task sets; and (3) Comparisons per task set: the number of comparisons per task set that each algorithm takes. When interarrival times are small, more containing relations among tasks are likely to happen. Figure 12 shows that the heuristic algorithms perform worse under this condition and tend to reject more tasks, especially when the computation time ratio is larger. Set-Scheduler always reaches 100% acceptance rate since it is an optimal scheduler. In the figure, because the characteristics of the data with different standard deviation ratios are similar, only the data with standard deviation ratio equal to 0.8 are depicted. When success ratio is concerned, which can be seen in Figure 4, the heuristic algorithms performs even worse. Generally speaking, the heuristic algorithms can usually produce suboptimal schedules, but fail to produce the optimal ones most of the time. The search space is shown in Figure 14. Set-Scheduler performs well at the expense of the complexity, which may become very large when the interarrival times are small. The cost is more reasonable while the interarrival times between tasks are not too small.
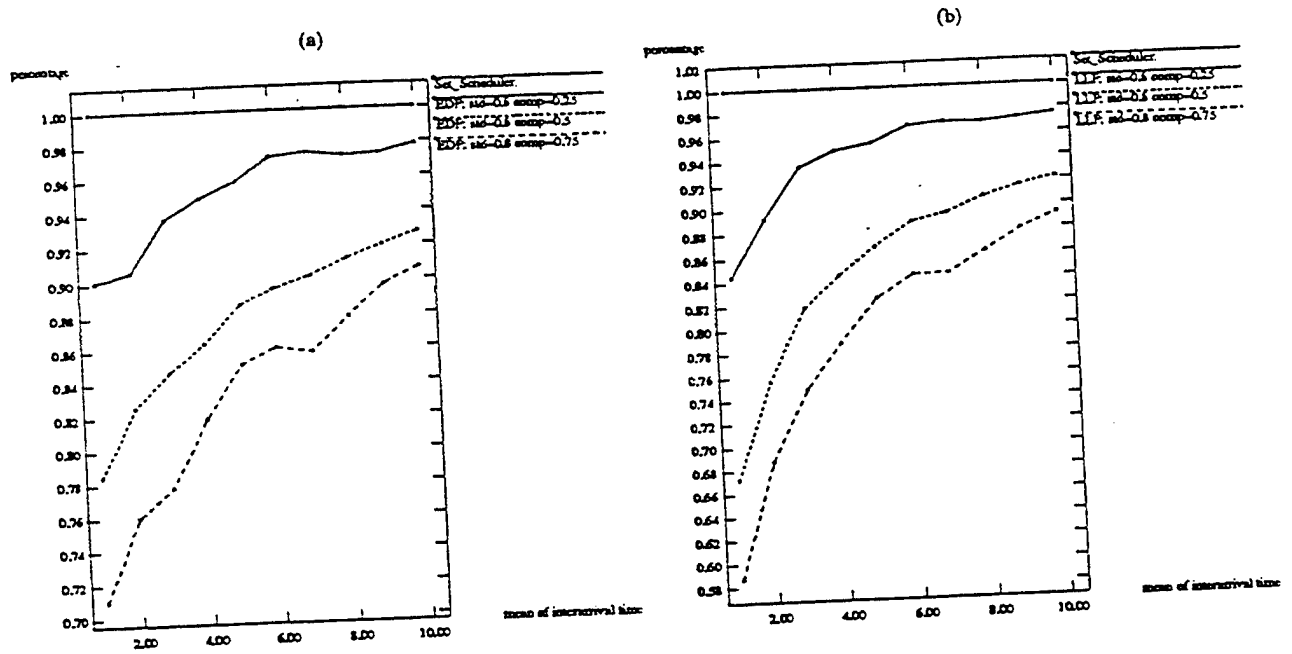
251

Figure 12: Percentage of accepted tasks (a) EDF (b) LLF compared with Set-Scheduler
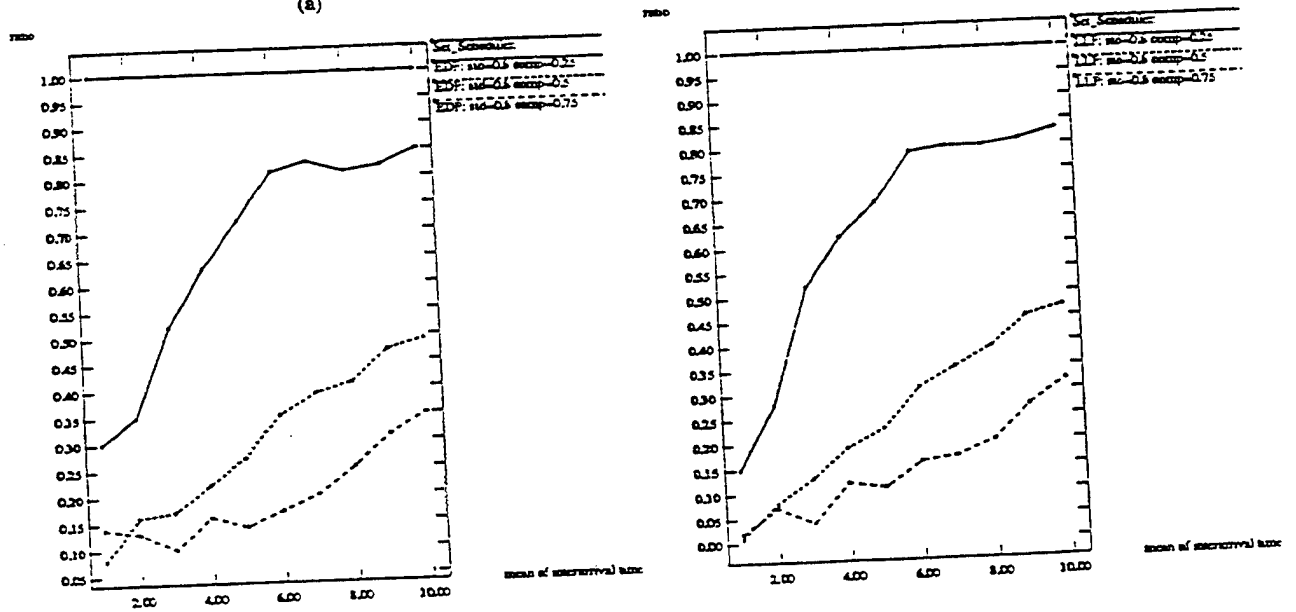

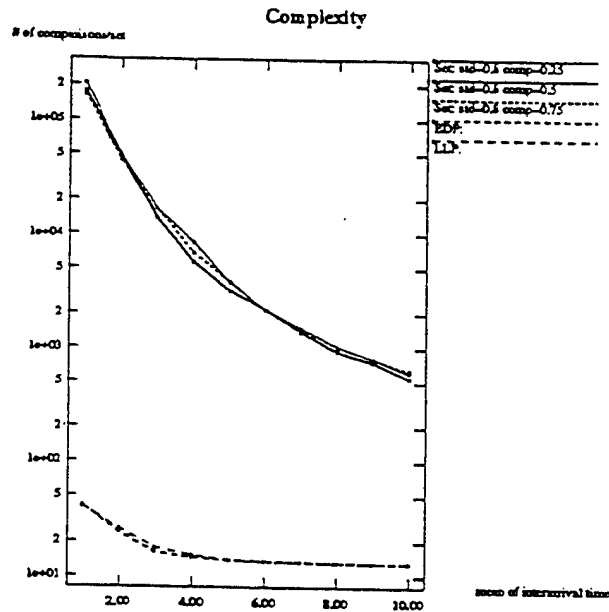Figure 13: Success Ratio (a) EDF (b) LLF compared with Set-Scheduler

Figure 14: Number of comparisons per task set

5 Conclusion Remarks and Future Work

In this paper, we discuss the optimization techniques in real-time scheduling for aperiodic tasks in a uniprocessor system with the non-preemptive discipline. We first propose a Sequence-Scheduler algorithm to compute the optimal schedule for a sequence in $O(n^2)$ time. Then a Set-Scheduler algorithm is proposed based on the super sequence and Sequence-Scheduler algorithm. The complexity of our Set-Scheduler algorithm is $O(N * n^2)$, compared to $O(N * n)$ for the feasibility test by Erschler *et al.*, where N might be as large as exponential in the worst case. However, our simulation results show that the cost is reasonable for the average case. We explore the temporal properties concerning the optimization issues, and present several theorems to formalize the results. The study of temporal properties on a uniprocessor may serve as a base for the more complex cases in multiprocessor systems.

For the future work, we propose to incorporate the decomposition technique [18] into our scheduling algorithm. Under this approach a task set can be decomposed into subsets, which results in backtracking points to reduce the search space. This has been shown to be

useful in reducing the search space substantially when the task set is well decomposable.

References

[1] M. Dertouzos. Control robotics: The procedural control of physical processes. In *Proceedings of the IFIP Congress*, pages 807–813, 1974.

[2] J. Erschler, G. Fontan, C. Merce, and F. Roubellat. A new dominance concept in scheduling n jobs on a single machine with ready times and due dates. *Operations Research*, 31(1):114–127, Jan. 1983.

[3] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman Company, San Francisco, 1979.

[4] D. W. Gillies and J. W-S. Liu. Greed in resource scheduling. In *IEEE Real-Time Systems Symposium*, pages 285–294, Dec. 1989.

[5] O. Gudmundsson, D. Mosse, K.T. Ko, A.K. Agrawala, and S.K. Tripathi. Maruti: A platform for hard real-time applications. In *Workshop on Operating Systems for Mission Critical Computing*, pages C1–C14, Sep. 1989.

[6] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault tolerant real-time systems: The mars approach. *IEEE Micro*, 9(1):25–40, Feb. 1989.

[7] J. P. Lehoczky. Fixed priority scheduling of periodic tasks with arbitrary deadlines. In *IEEE Real-Time Systems Symposium*, pages 201–209, Dec. 1990.

[8] J.Y. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. *Performance Evaluation*, 2(4):237–250, 1982.

[9] S.T. Levi, S.K. Tripathi, S.D. Carson, and A.K. Agrawala. The maruti hard real-time operating system. *ACM SIGOPS, Operating Systems Review*, 23:90–106, July 1989.

[10] C. L. Liu and J. Layland. Scheduling algorithm for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, Jan. 1973.

[11] G. McMahon and M. Florian. On scheduling with ready times and due dates to minimize maximum lateness. *Operations Research*, 23(3):475–482, May 1975.

[12] A. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. PhD thesis, MIT Laboratory for Computer Science, May 1983.

[13] Manas Saksena and Ashok Agrawala. Temporal analysis for static hard-real time scheduling. In *Proceedings 12th International Phoenix Conference on Computers and Communications*, pages 538–544, March 1993.

[14] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sep. 1990.

[15] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. Technical Report CMU-CS-87-181, Department of Computer Science, Carnegie-Mellon University, 1987.

[16] J.A. Stankovic and K. Ramamritham. The spring kernel: Operating system support for critical, hard real-time systems. In *Workshop on Operating Systems for Mission Critical Computing*, pages A1–A9, Sep. 1989.

[17] J. Xu and D. L. Parnas. Scheduling processes with release times, deadlines, precedence, and exclusion relations. *IEEE Transactions on Software Engineering*, SE-16(3):360–369, March 1990.

[18] X. Yuan and A. K. Agrawala. A decomposition approach to nonpreemptive scheduling in hard real-time systems. In *IEEE Real-Time Systems Symposium*, Dec. 1989.

[19] W. Zhao, K. Ramamritham, and J. A. Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Transactions on Computers*, C-36(8):949–960, Aug. 1987.

[20] W. Zhao, K. Ramamritham, and J. A. Stankovic. Scheduling tasks with resource requirements in a hard real-time system. *IEEE Transactions on Software Engineering*, SE-13(5):564–577, May 1987.

REPORT DOCUMENTATION PAGE

Form approved
OMB No 074-0188

1. **AGENCY USE ONLY (leave blank)**	2. **REPORT DATE** January 1994	3. **REPORT TYPE END DATES COVERED** Technical Report

4. TITLE AND SUBTITLE
Optimization in Non-Preemptive Scheduling for Aperiodic Tasks

5. FUNDING NUMNBERS

N00014-91-C-0195
DSAG-60-92-C-0055

6. AUTHOR(S)
Shyh-In Hwang, Sheng-Tzong Cheng and Ashok K. Agrawala

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Maryland
Department of Computer Science
A.V. Williams Building
College Park, MD 20742

8. PERFORMING ORGANIZATION REPORT NUMBER
CS-TR-3216
UMIACS-TR-94-14

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Honeywell Inc.
3600 Technology Drive
Minneapolis, MN 55148

Phillips Laboratory
Directorate of Contracting
3651 Lowry Avenue, SE
Kirtland AFB, NM 87117-5777

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12.a. DISTRIBUTION/ AVAILABILITY STATEMENT

12.b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Real-time computer systems have become more and more important in many applications, such as robot control, flight control, and other mission-critical jobs. The correctness of the system depends on the temporal correctness as well as the functional correctness of the tasks. We propose a scheduling algorithm based on an analytic model. Our goal is to derive the optimal schedule for a given set of aperiodic tasks such that the number of rejected tasks is minimized, and then the finish time of the schedule is also minimized. The scheduling problem with a nonpreemptive discipline in a uniprocessor system is considered. We first show that if a total ordering is given, this can be done in $O(n2)$ time by dynamic programming technique, were n is the size to the task set. When the restriction of the total ordering is released, it is known to be NP-complete [3]. We discuss the super sequence [18] which has been shown to be useful in reducing the search space for testing the feasibility of a task set. By extendng the idea and introducing the concept of conformation, the scheduling process can be divided into two phases: computing the pruned search space and computing the optimal schedule for each sequence in the search space. While the complexity of the algorithm in the worst case remains exponential, our simulation results show that the cost is reasonable for the average case.

14. SUBJECT TERMS
Operating Systems Process, Process Management
Analysis of Algorithms and Problem Complexity, Nonnumerical Algorithms and Problems

15. NUMBER OF PAGES
44

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlisted

MSN 7540-01 280-5500

257

Standard Form 298 (Rev 2-89)

A Decomposition Approach to
Nonpreemptive Real-Time
Scheduling*

Xiaoping (George) Yuan[†] and Ashok K. Agrawala
Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

Let us consider the problem of scheduling a set of n tasks on a single processor such that a feasible schedule which satisfies the time constraints of each task is generated. It is recognized that an exhaustive search may be required to generate such a feasible schedule or to assure that there does not exists one. In that case the computational complexity of the search is of the order $n!$.

We propose to generate the feasible schedule in two steps. In the first step we decompose the set of tasks into m subsets by analyzing their ready times and deadlines. An ordering of these subsets is also specified such that in a feasible schedule all tasks in an earlier subset in the ordering appears before tasks in a later subset. With no simplification of scheduling of tasks in a subset, the scheduling complexity is $O(\sum_{i=1}^{m} n_i!)$, where n_i is the number of tasks in the ith subset.

The improvement of this approach towards reducing the scheduling complexity depends on the the number and the size of subsets generated. Experimental results indicates that significant improvement can be expected in most situations.

Contents

I Introduction

Consider the problem of nonpreemptive scheduling of n tasks on a single CPU of a hard real-time system. For task T_i, identified as i, the scheduling request consists of a triple $< c_i, r_i, d_i >$ where c_i is the computation time, r_i the ready time before which task i can not start, and d_i the deadline before which the computation must be completed. Time interval $[r_i, d_i]$ is called the *time window* denoted by w_i. The window length $|w_i|$ is $d_i - r_i$. In a hard real-time system, a schedule is called *feasible* if all tasks are processed within their individual windows.

The result of the scheduling process is a schedule in which for any task i, a start time s_i and a finish time f_i is identified, where $f_i = s_i + c_i$. Clearly, a schedule is feasible, if for every task i,

$$r_i \le s_i \le d_i - c_i. \tag{1}$$

The scheduling process is not preemptive only if for any two tasks i and j,

$$s_i \le s_j \Rightarrow s_i + c_i \le s_j. \tag{2}$$

In other words, when task i is scheduled, a span of nonpreemptable processing time, c_i, is allocated for it. No other task may be in execution during that time span. Thus the scheduling problem is to find a mapping from a task set $\{i\}$ to a start time set $\{s_i\}$, such that constraints in (1) and (2) are met. Note that for a given set of tasks $\{i\}$, there may be none, one or many feasible schedules.

In general the nonpreemptive real-time scheduling problem is known to be NP-complete [Gare79]. To find a feasible schedule, the number of schedules to be examined is $O(n!)$, which we count as the *scheduling complexity*. Heuristic techniques can be used [Ma84, McMa75, Mok83, Zhao87] to reduce the complexity. This reduction, however, is achieved at the cost of obtaining a potentially sub-optimal solution. That is, when looking for feasible schedules, heuristic techniques may not yield a feasible schedule, even though one exists. Schedules based on the earliest-deadline-first, or minimum-laxity-first rules are examples of such heuristics used in scheduling.

An alternate approach is to develop analytical methods for scheduling [Ersc83, Liu73]. This approach analyzes the relationships among real-time tasks and schedules. The purpose is to precisely determine optimal task schedules, or narrow the search scope from the original search space.

The objective of this research is to develop *correct* and efficient algorithms for nonpreemptive real-time scheduling. We call a scheduling algorithm *correct*, if whenever a feasible schedule exists, the algorithm can find it.

261

In this paper, we present an analytical decomposition approach for real-time scheduling. The strategy is to divide a set of tasks into a sequence of subsets, such that the search for feasible schedules is only performed within each subset. The decomposition technique used for generating the sequence of subsets assures that in a feasible schedule all tasks in a subset earlier in the sequence are scheduled before any task in a later subset. Backtracking in the search is bounded within each subset, which significantly reduces the scheduling complexity.

There are several different strategies which can be used to subset tasks. The decomposition strategy discussed in this paper is to use a relation called the *leading relation* which depends on the tasks' relative window positions.

We performed an experiment which examined the number and size of subsets with regard to the number of tasks, task arrival rate, and window length. We found that, in general, the number of tasks in any subset is independent of the total number of tasks to be scheduled, if the task window lengths are bounded. The decomposition scheduling is a polynomial computation. As a consequence, the decomposition method is very practical for the implementation.

In section II we present some basic notions used in the paper. In section III we discuss a case where all the tasks have the leading relation with each other. Our approach of decomposition scheduling is introduced in section IV along with concepts of the single schedule subset and decomposed leading schedule sequence. We present our experiment results in section V. Our conclusion and future research in section VII.

II Background

If we consider any two tasks i and j, they must have one of these three relations:

1. *leading* - $i \prec j$ (or $j \prec i$), where if $r_i \leq r_j$, $d_i \leq d_j$ and $w_i \neq w_j$.

2. *matching* - $i \| j$, if $r_i = r_j$ and $d_i = d_j$.

3. *containing* - $i \sqcup j$ (or $j \sqcup i$), if $r_i < r_j$ and $d_j < d_i$.

These three relations are shown in Fig. 1. It is easy to see that the leading, matching and containing relations are all transitive. Additionally, if $i \| j$ or $i \sqcup j$, we say that i and j are *concurrent*.

A *length* is associated with a schedule which is the finish time of the last task in the schedule. One example is shown in Fig. 2.

The concept of *dominance* was introduced in [Ersc83], and we will use it later in the discussion.

Definition 1 For two schedules S_1 and S_2, S_1 *dominates* S_2 if and only if:

$$S_2 \; feasible \Rightarrow S_1 \; feasible.$$

262
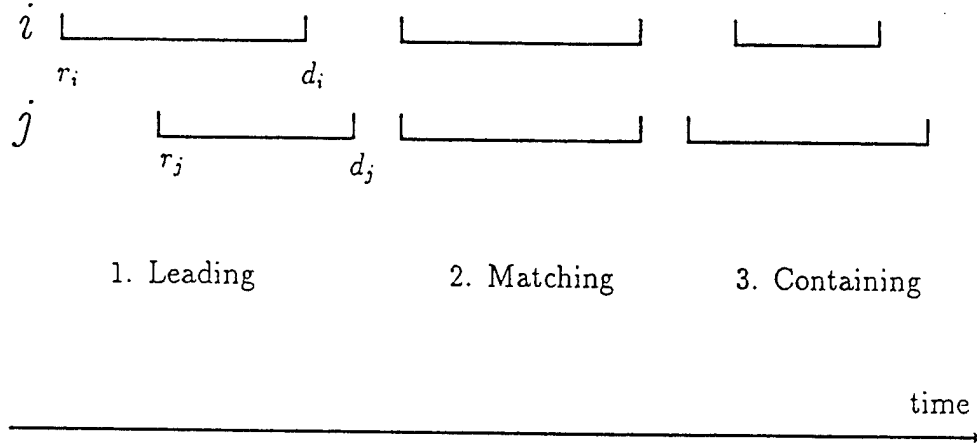
Figure 1: Task window relations

Definition 2 A set of schedules, S, is *dominant* if $\forall S_2 \notin S$, $\exists S_1 \in S$ such that S_1 dominates S_2.

A schedule is *dominant* if it dominates all other schedules.

III   The Leading Schedule Sequence

Let us consider the case where for a set of task $\{i\}$, every pair of tasks in this set has a leading relation, i.e. $i \prec j$ or $j \prec i$, for every $i, j, i \neq j$.

Based on the leading relation we can define a total order of tasks for the set. We define the *leading schedule sequence* (LSS) to be a sequence of tasks in which tasks are in order according to the leading relation, that is, for any i and j, $i \xrightarrow{\cdot} j \iff i \prec j$, where $i \xrightarrow{\cdot} j$ means that i is scheduled in front of j.

Theorem 1 For a set of tasks all of which have a pairwise leading relation, the schedule where tasks are sequenced in order of the leading schedule sequence is a dominant one.

Proof: We prove this theorem by construction.
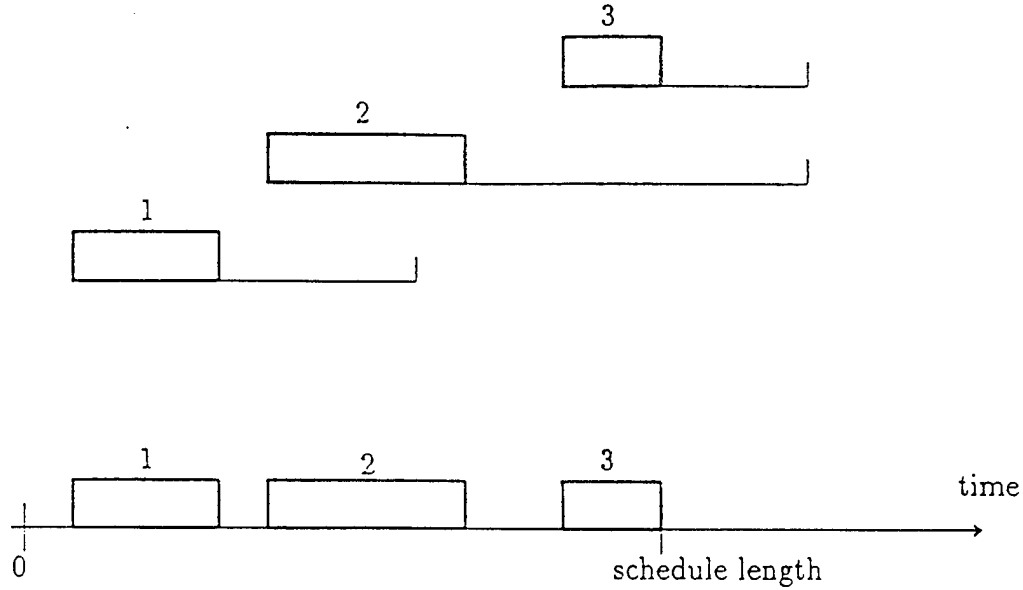
Figure 2: An example of one schedule

Suppose this set of tasks has a feasible schedule S in which tasks occur in a different sequence than the leading schedule sequence. When we examine this schedule, let i and j be the first pair of tasks that are not ordered by the leading relation, i.e. $i \prec j$, but $j \rightharpoonup i$. From the leading relation we know that $r_i \leq r_j$ and $d_i \leq d_j$.

Since j and i are the first such pair, deadlines of all tasks between j and i in S must be greater than or equal to d_j as well as d_i. In S, let us construct another schedule S' by moving i from the current position in S to the position just in front of j. The start time and finish time of tasks between j and i including j will be increased by no more than c_i. And so, no task between j and i including j will finish later than d_i. Meanwhile, the rest of this schedule is unchanged. Thus if S is feasible, the new schedule S' will be feasible too.

By repeating the process of constructing S' from S, we obtain a schedule which has all tasks ordered according to the leading relation, such that if the original schedule is feasible, so is the constructed one. □

Thus if there exists a set of feasible schedules, the set must contain schedules that are conforming to the order of the leading schedule sequence. The result can be generalized to the situation where there exist matching windows. The generalization is to combine

the tasks with the same window into one task whose computation time is the sum of the computation times of all these tasks.

We will see that the above leading schedule sequence is a special case of the decomposed leading schedule sequence introduced in the next section.

IV Task Decomposition

A. Philosophy

To solve the general real-time scheduling problem with n tasks, the number of schedules to be examined can be as much as $O(n!)$. However, taking a closer look, we find that every task has an important property called the *locality* of a task, that is, a task is time-bounded by its time window. Furthermore, if any two task windows are not overlapping, there is only one possible order for them. The above facts motivate us to separate the tasks into subsets according to their different time localities.

The *decomposition scheduling* can be divided into two steps: decomposition and scheduling.

First, a set of n tasks is decomposed into a sequence of m subsets such that the orders of subsets are fixed. The order of a task is determined only relative to the other tasks within its own subset. The sequence of the subsets is called the *decomposed schedule sequence.* The decomposition should be so developed that the schedulability of tasks is not damaged at all. The decomposition by using the leading relation introduced in this paper shows this property.

The second step is to schedule the subsets in the sequence order. It always selects a schedule for each subset with the shortest length, so that when a subset is scheduled, the time span available for it is maximized. In this way, the total number of schedules to be examined is only $O(\sum_{i=1}^{m} n_i!)$, where n_i is the number of tasks in the ith subset ($\sum_{i=1}^{m} n_i = n$).

The only remaining problem is how to decompose a set of tasks into a sequence of subsets of tasks such that a feasible schedule is guaranteed to be found if one exists. In the rest of this paper, we outline how to use the leading relation as a means to divide the task set.

B. Decomposition Scheme

A set of tasks is called the *single schedule subset* (sss), represented as τ, if

$$\forall i \in \tau \; \exists j \in \tau \; (i \sqcup j) \vee (j \sqcup i) \vee (i \| j).$$

In other words, each task window is contained in the window of another task, contains the window of another task, or matches the window of another task in the subset.

Given a set of tasks $\{i\}$, we can decompose it into a sequence of single schedule subsets $\tau^1, \tau^2, \cdots, \tau^k$ such that all the tasks in τ^i are leading to all the tasks in τ^{i+1}.

The *decomposed leading schedule sequence* ($DLSS$) is defined to be a sequence of single schedule subsets, denoted as:

$$DLSS = \tau^1 \circ \tau^2 \circ \cdots \circ \tau^m,$$

such that $\forall k^i \in \tau^i \; \forall k^j \in \tau^j \; k^i \prec k^j$, for $1 \leq i < j \leq m$, (denoted as $\tau^i \prec \tau^j$), and τ^i can not be further decomposed, for $i = 1, \cdots, m$. Symbol $\circ$ represents a concatenating operation.

Note that if a task in τ^i does not lead another task in τ^j for $i < j$, they must have a matching or containing relation. If this happens, τ^i and τ^j can not be different single schedule subsets. Clearly, all n tasks may belong to a single schedule subset.

Theorem 2 The set of schedules conforming to the decomposed leading schedule sequence is dominant.

Proof: Assume that if there are two tasks $k^i \in \tau^i$ and $k^j \in \tau^j$, where $\tau^i \prec \tau^j$. There is no common concurrent task with both k^i and k^j. k^j is positioned in front of k^i in a feasible schedule (S). Specifically, $S = (\cdots) \circ (k^j \circ \cdots k^i) \circ (\cdots)$. Let us define $S' = (k^j \circ \cdots k^i)$ for abbreviation ($S = (\cdots) \circ S' \circ (\cdots)$). The new schedule created by exchanging k^i's position with k^j's is still feasible.

Without loss of generality, suppose that k^i and k^j are the first such pair in S. Tasks between k^j and k^i are led by k^j, or concurrent with k^j, but not leading to and not concurrent with k^i. Since $k^i \prec k^j$ (i.e. $\tau_i \leq \tau_j$), switching k^i's and k^j's positions will not increase the finish time of S', which is defined as the finish time of the last task in S'. All the tasks between k^i and k^j, including k^j, are led by k^i, i.e. having deadlines greater than or equal to d_{k^i}. If S is feasible with k^i as the last task in S', it will be still feasible after the switching. $\quad\square$

Note that if the set of schedules that are following the decomposed leading schedule sequence is *empty*, there is no feasible schedule available for the tasks to be scheduled.

C. Decomposition Algorithm

Decomposing a set of tasks into single schedule subsets, the algorithm starts with the tasks having been sorted by their ready times (using their deadlines if their ready times are the same).

The algorithm uses one single loop to determine which single schedule subset the current task should belong to. The loop consists of two parts. The first part is a *while* loop which merges single schedule subsets into one, if the current task is contained by them. The second part decides whether the current task can form a new single schedule subset, or join with another single schedule subset.

The Leading-relation Decomposition Algorithm

begin

       /* Initialization. */

       $k = 1; \tau^1 = \{1\}$;

       $\tau_{\tau^1} = \tau_1; d_{\tau^1} = d_1$;

       for $i = 2$ to n do /* Go over the task list. */

           $l = k - 1$;   /* l is the index of single schedule subsets. */

           $continue = \text{TRUE}$;

           while $(l > 0) \wedge (continue)$ do

                /* Merge single schedule subsets if the current task is concurrent

                  with tasks in different subsets. */

                if $(d_{\tau^l} > d_i)$

                    $\tau^l = \tau^k \cup \tau^l$;

                    $d_{\tau^l} = d_{\tau^k}$;

                    $k = l$;

                else

                  $continue = \text{FALSE}$;

                $l = l - 1$;

           od

           if $(\tau_{\tau^k} \leq \tau_i) \wedge (d_i < d_{\tau^k})$

                /* The current task is concurrent with tasks in the current subset.*/

                $\tau^k = \tau^k \cup \{i\}$;

           else if $(\tau_{\tau^k} \leq \tau_i) \wedge (d_{\tau^k} \leq d_i)$

                /* The current task is led by all the tasks in the current subset.

                  A new single schedule subset is created only containing the current task.*/

                $k = k + 1$;

                $\tau^k = \{i\}$;

                $\tau_{\tau^k} = \tau_i$;

                $d_{\tau^k} = d_i$.

       od

   end

In this algorithm, the outer loop is executed n times. The *while* loop is executed no more than the number of time proportional to n in total, since no more than n subsets can be merged during the whole execution of the algorithm with n tasks. Thus the complexity of this algorithm is only $O(n)$. If we count in the sorting complexity, the decomposition will cost no more than $O(n \log n)$.

D. Scheduling Scheme

After tasks has been decomposed into a sequence of subsets, scheduling should be performed on each subset in the sequence order, such that the schedule on each subset is of the shortest length. A brute force method is to give an exhaustive search whose computational complexity amounts to $O(n_i!)$, where n_i is the number of tasks in the ith subset.

In [Yuan89b], other scheme is explored for scheduling a subset. The method is to first build a super-sequence where tasks may have several occurrences. The occurrence of a task is decided by its relative window position in the subset. Selecting one occurrence for every task in the super-sequence forms a schedule. A complete search costs $O(n_i^{\frac{1}{2}(n_i - n_i^{\frac{1}{2}})})$ in the worst case. When we made a few calculation samples of $n_i^{\frac{1}{2}(n_i - n_i^{\frac{1}{2}})}$ with n_i less than 100, $n_i^{\frac{1}{2}(n_i - n_i^{\frac{1}{2}})}$ is a much smaller number than $n_i!$, as shown in the cited paper.

Since the set of schedules following the decomposed leading schedule sequence is dominant, and since the subsets are scheduled in the sequence order with their shortest length, it is proved that the decomposition scheduling with the leading relation is correct [Yuan89b].

V Empirical Study

A. Experiment

In order to observe the behavior of the number of tasks in a single schedule subset and number of the subsets to be created with regard to the number of tasks to be scheduled, task arrival rate, and task window length, we conduct an experiment as an example to see the feasibility of our approach for practical implementation.

The outputs we are interested in are:

1. the number of single schedule subsets (sss),

2. the number of window concurrences,

3. the maximum number of tasks in single schedule subsets,

4. the minimum number of tasks in single schedule subsets, and

5. the average number of tasks in single schedule subsets.

One window concurrence is counted for any two tasks i and j if i and j have a concurrent relation. We call the number of tasks in a single schedule subset as the *size* of the subset.

Meanwhile, we change the following parameters independently to watch the changes in the outputs,

1. the number of total tasks,

2. task arrival rate, and

3. window length.

The data is shown in Table 1-4[1] in the end of this paper. Following are basic rules in the experiment.

1. The computation time is uniformly distributed over $(0, \alpha]$.

2. The task interarrival is uniformly distributed over $[0, \beta)$. The arrival rate is $2/\beta$.

3. The window length is also randomly created by controlling the laxity for each task. The laxity of a task is the difference between its window length and its computation time. The laxity is uniformly distributed $[0, \gamma)$. The distribution guarantees the window length greater than the computation time for the task.

We notice that the arrival rate should be less than or equal to the service rate, otherwise, there are congestions in the system, which will result in deadline-missing. In other words, $2/\beta \leq 2/\alpha$. That is,

$$\alpha \leq \beta.$$

The random numbers are provided by function $drand()$ in the UNIX operating system. The numbers are uniformly distributed over $[0, 1)$ [Stev86].

In the experiment, we found that the minimum size of single schedule subsets is always one.

B. The Result Explanation and Observation

From the experiment results , we found that when the average window length increases (γ increases), the number of single schedule subsets reduces and the maximum size of single schedule subsets slightly increases. The result is expected, since the larger some task

---

[1] In the tables, *number* is represented by num. *Window* by W. *Concurrences* by concurr. *Average* by avg. The *Single schedule subset* by sss.

windows are, the more tasks may be concurrent with them. These tasks may be in the same single schedule subset.

When β increases, that is, the arrival rate decreases, the number of single schedule subsets increases, and maximum size of single schedule subsets decreases. The result is also expected, since when the arrival rate decreases, the opportunity of tasks concurrent with each other decreases too. Most tasks have the leading relation with each other.
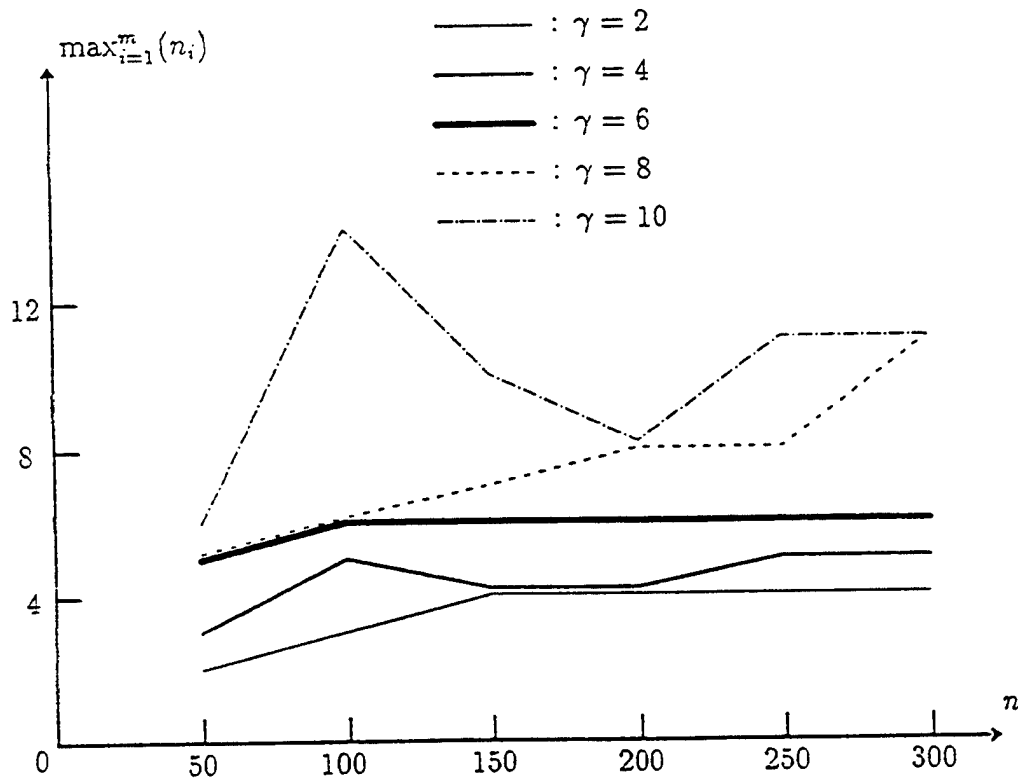
Figure 3: The relationship between the size of single schedule subsets and the number of tasks with regard to the laxity parameter γ, where $\alpha = 4$, $\beta = 4$.

Fig. 3 shows the relationship between the maximum size of single schedule subsets and the number of tasks to be scheduled. From the experiment, we found that the size of a single schedule set never exceeds 14 even when there are 300 tasks being scheduled. The observation indicates that for most cases $max_{i=1}^{m}(n_i)$ is a constant.

We show the relationship between the number of subsets (m) and number of tasks to be scheduled in Fig. 4 and Fig. 5 with regard to different window length and arrival rate distributions.
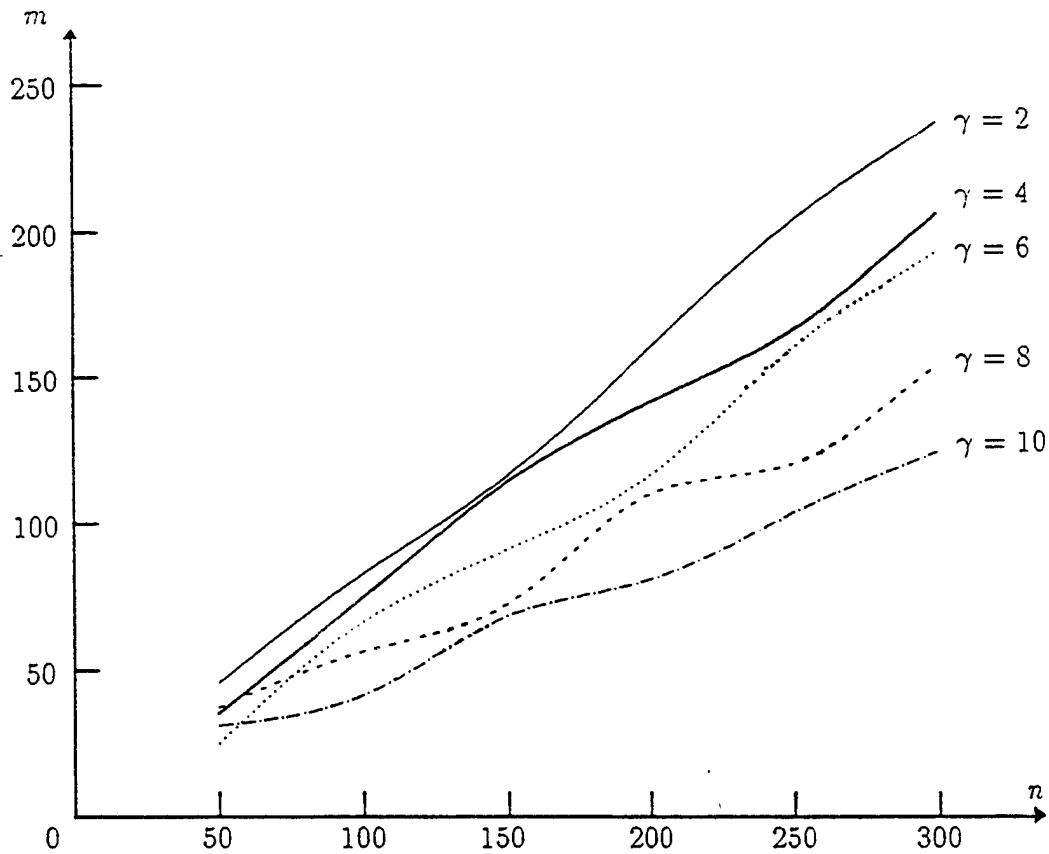
Figure 4: The relationship between the number of single schedule subsets and the number of tasks with regard to the laxity parameter γ, where $\alpha = 4$, $\beta = 4$.

VI   Final Remarks

In this paper, we examine the problem of nonpreemptive scheduling of n tasks on a single CPU in hard real-time systems. We propose a correct decomposition strategy for the scheduling. The strategy significantly reduces the scheduling complexity for most cases.

In this paper we have examined a decomposition technique based only on the windows of tasks. By taking into account the computation time requirements, the decomposition can be made stronger [Yuan89a]. The decomposition approach may also be extended to consider precedence and other dependences among tasks. This aspect of decomposition technique needs further study.

Acknowledgements

Thanks to S. Mukherjee for his help in coding the decomposition algorithm for our
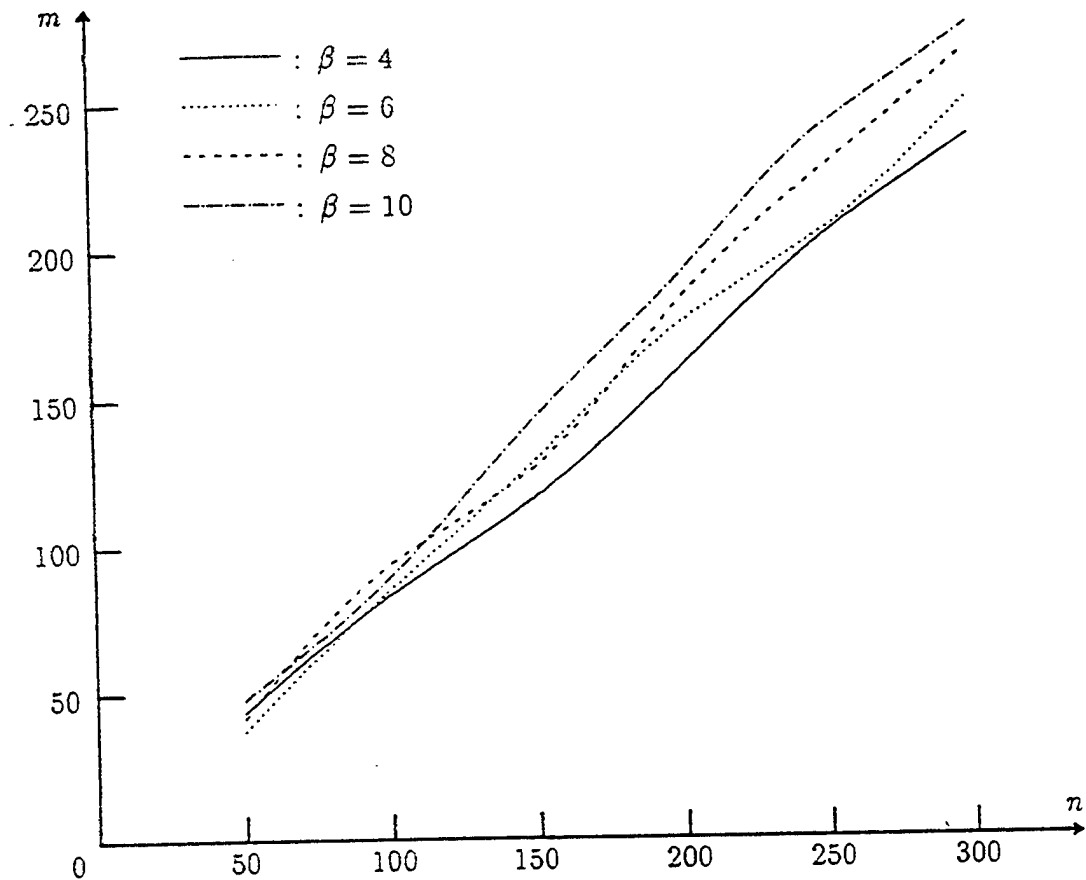
Figure 5: The relationship between the number of single schedule subsets and the number of tasks with regard to the arrival rate parameter β, where $\alpha = 4$, $\gamma = 2$.

experiment.

References

[Ersc83] Erschler, J., Fontan, G., Merce, C., and Roubellat, F., "A New Dominance Concept in Scheduling n Jobs on a Single Machine with Ready Times and Due Dates", *Operations Research*, Vol. 31, No. 1, pp. 114–127, Jan. 1983.

[Gare79] Garey, M. R. and Johnson, D. S., *Computers and Intractability, a Guide to the Theory of NP-Completeness*, W. H. Freeman Company, San Francisco, 1979.

[Liu73] Liu, C. L. and Layland, J., "Scheduling Algorithm for Multiprogramming in a Hard Real-Time Environment", *Journal of the ACM*, Vol. 20, pp. 46–61, Jan.

1973.

[Ma84]   Ma, P. R., "A Model to Solve Timing-Critical Application Problems in Distributed Computing Systems", *IEEE Computer*, Vol. 17, pp. 62–68, Jan. 1984.

[McMa75] McMahon, G. and Florian, M., "On Scheduling with Ready Times and Due Dates to Minimize Maximum Lateness", *Operations Research*, Vol. 23, No. 3, pp. 475–482, May 1975.

[Mok83]   Mok, A. K., "Fundamental Design Problems for the Hard Real-time Environments", May 1983, MIT Ph.D. Dissertation.

[Stev86]   Steven V. Earhart, Ed., *UNIX Programmer's Manual: System Calls and Library Routines*, Volume Volume 2, CBS College Publishing, 1986.

[Yuan89a] Yuan, X. and Agrawala, A. K., *Decomposition with a Strongly-Leading Relation for Hard Real-Time Scheduling*, Technical Report to be Published, Dept. of Computer Science, Univ. of Maryland, Coll. Pk., MD 20742. Mar. 1989.

[Yuan89b] Yuan, X. and Agrawala, A. K., *Scheduling Real-Time Task in Single Schedule Subsets*, Technical Report to be Published, Dept. of Computer Science, Univ. of Maryland, Coll. Pk., MD 20742, Mar. 1989.

[Zhao87] Zhao, W., Ramamritham, K., and Stankovic, J. A., "Scheduling Tasks with Resource requirements in a Hard Real-Time System", *IEEE Trans. on Soft. Eng.*, Vol. SE-13, No. 5, pp. 564–577, May 1987.

num of tasks	γ	avg W. length	num of concurr.	num of sss	sss size max	sss size avg
	2	3.02	5	45	2	1.11
	4	4.33	18	35	3	1.43
50	6	5.08	27	25	5	2.00
	8	6.53	14	36	5	1.39
	10	7.03	26	30	6	1.67
	2	3.11	18	82	3	1.22
	4	4.18	28	74	5	1.35
100	6	4.34	37	68	6	1.47
	8	5.84 ·	56	55	6	1.82
	10	7.01	77	41	14	2.44
	2	3.09	32	118	4	1.27
	4	3.78	40	114	4	1.32
150	6	4.93	68	92	6	1.63
	8	5.85	91	73	7	2.05
	10	6.56	111	68	10	2.21
	2	3.11	41	159	4	1.26
	4	3.67	64	144	4	1.39
200	6	4.94	98	116	6	1.72
	8	5.94	102	113	8	1.77
	10	7.27	161	81	8	2.47
	2	3.02	46	204	4	1.23
	4	3.82	93	167	5	1.50
250	6	4.83	107	160	6	1.56
	8	6.33	162	121	8	2.07
	10	7.29	186	103	11	2.43
	2	2.98	66	237	4	1.27
	4	3.90	105	205	5	1.46
300	6	5.09	125	194	6	1.55
	8	6.21	179	153	11	1.96
	10	6.92	228	125	11	2.40

Table 1: $\alpha = 4$, $\beta = 4$

num of tasks	γ	avg W. length	num of concurr.	num of sss	sss size max	sss size avg
	2	2.79	13	38	3	1.32
	4	4.22	7	43	3	1.16
50	6	4.45	16	36	3	1.39
	8	6.21	21	31	4	1.61
	10	6.82	23	30	5	1.67
	2	2.96	16	85	3	1.18
	4	3.93	16	85	3	1.18
100	6	3.93	16	85	3	1.18
	8	6.32	31	73	4	1.37
	10	7.21	52	58	7	1.72
	2	3.29	20	130	4	1.15
	4	3.98	29	125	5	1.20
150	6	5.33	42	111	4	1.35
	8	5.90	55	100	5	1.50
	10	6.96	77	83	8	1.81
	2	3.04	25	175	4	1.14
	4	3.90	48	153	3	1.31
200	6	5.18	55	148	9	1.35
	8	5.95	87	128	8	1.56
	10	7.18	84	128	6	1.56
	2	2.93	45	206	3	1.21
	4	4.08	55	200	6	1.25
250	6	5.10	55	197	5	1.27
	8	6.15	78	175	8	1.43
	10	6.78	98	162	7	1.54
	2	3.00	54	250	4	1.20
	4	4.13	47	254	5	1.18
300	6	4.97	67	236	4	1.27
	8	5.82	96	214	5	1.40
	10	6.97	161	159	8	1.89

Table 2: $\alpha = 4$, $\beta = 6$

num of tasks	γ	avg W. length	num of concurr.	num of sss	sss size max	sss size avg
			Table 3: $\alpha = 4$, $\beta = 8$			
	2	3.24	9	42	3	1.19
	4	3.56	5	45	3	1.11
50	6	4.67	11	40	3	1.25
	8	5.95	3	47	2	1.06
	10	6.51	20	32	5	1.56
	2	2.99	6	94	2	1.06
	4	4.20	13	87	3	1.15
100	6	5.04	26	75	3	1.33
	8	5.77	29	73	5	1.37
	10	6.91	33	70	5	1.43
	2	2.99	23	127	4	1.18
	4	4.01	28	126	4	1.19
150	6	5.27	28	122	3	1.23
	8	6.00	32	120	5	1.25
	10	7.22	39	112	6	1.34
	2	3.09	19	182	3	1.10
	4	3.94	18	182	3	1.10
200	6	4.87	31	171	4	1.17
	8	5.91	51	154	5	1.30
	10	6.96	78	130	6	1.54
	2	2.91	23	227	3	1.10
	4	4.01	48	206	4	1.21
250	6	5.01	52	204	5	1.23
	8	5.84	56	195	5	1.28
	10	7.48	60	191	4	1.31
	2	2.99	35	267	4	1.12
	4	4.05	30	270	3	1.11
300	6	5.07	60	241	4	1.24
	8	5.99	63	240	4	1.25
	10	7.13	110	199	6	1.51

num of tasks	γ	avg W. length	num of concurr.	num of sss	sss size max	sss size avg
			Table 4: $\alpha = 4$, $\beta = 10$			
50	2	3.44	2	48	2	1.04
	4	4.11	5	45	3	1.11
	6	5.50	4	46	3	1.09
	8	6.69	11	41	4	1.22
	10	6.61	15	38	5	1.32
100	2	3.02	--	89	4	1.12
	4	4.01	14	87	4	1.15
	6	4.95	21	80	4	1.25
	8	6.59	18	82	3	1.22
	10	6.68	25	76	5	1.32
150	2	3.00	6	144	2	1.04
	4	3.98	11	139	3	1.08
	6	4.72	11	139	2	1.08
	8	5.91	37	117	5	1.28
	10	7.10	43	109	6	1.38
200	2	3.00	18	182	3	1.10
	4	3.97	27	174	5	1.15
	6	5.19	23	177	3	1.13
	8	6.16	34	167	4	1.20
	10	6.90	52	153	4	1.31
250	2	2.94	24	228	3	1.10
	4	4.12	32	218	3	1.15
	6	5.10	43	213	4	1.17
	8	6.07	45	206	3	1.21
	10	6.96	51	200	4	1.25
300	2	3.05	27	274	3	1.09
	4	4.07	21	280	3	1.07
	6	5.04	54	251	6	1.20
	8	6.18	46	254	3	1.18
	10	6.61	61	243	8	1.23

Let us consider the problem of scheduling a set of n tasks on a single processor such that a feasible schedule which satisfies the time constraints of each task is generated. It is recognized that an exhaustive search may be required to generate such a feasible schedule or to assure that there does not exists one. In that case the computational complexity of the search is of the order $n!$.

We propose to generate the feasible schedule in two steps. In the first step we decompose the set of tasks into m subsets by analyzing their ready times and deadlines. An ordering of these subsets is also specified such that in a feasible schedule all tasks in an earlier subset in the ordering appears before tasks in a later subset. With no simplification of scheduling of tasks in a subset, the scheduling complexity is $O(\sum_{i=1}^{m} n_i!)$, where n_i is the number of tasks in the ith subset.

The improvement of this approach towards reducing the scheduling complexity depends on the the number and the size of subsets generated. Experimental results indicates that significant improvement can be expected in most situations.

REPORT DOCUMENTATION PAGE

Form approved
OMB No 074-0188

1. AGENCY USE ONLY (leave blank)	2. REPORT DATE Novemebr 1989	3. REPORT TYPE END DATES COVERED Technical Report

4. TITLE AND SUBTITLE A Decomposition Approach to Nonpreemptive Real-Time Scheduling	5. FUNDING NUMNBERS DASG60-87-X-0066

6. AUTHOR(S)

Xiaoping (George) Yuan, Ashok K. Agrawala, and Manas C. Saksena

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Maryland Department of Computer Science A.V. Williams Building College Park, MD 20742	8. PERFORMING ORGANIZATION REPORT NUMBER CS-TR-89-109 UMIACS-TR-2345

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES) US Army Strategic Defense Command Contr. & Acq. Mgt. Office CSSD-H-CRS, P.O. Box 1500 Huntsville, AL 35807-3801	10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12.a. DISTRIBUTION/ AVAILABILITY STATEMENT	12.b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Consider the problem of scheduling a set of n tasks on a uniprocessor such that a feasible schedule that satisfies each task's time constraints is generated. Traditionally, researchers have looked at all the tasks as a group and applied heuristic or enumeration search to it. We propose a new approach called the *decomposition scheduling* where tasks are decomposed into a sequence of subsets. Tasks are scheduled in each subset independently in the order of the sequence. It is proved that a feasible schedule can be generated as long as one exists for the tasks. In addition, the overall scheduling cost is reduced to the sum of the scheduling costs of the tasks in each subset.

Simulation experiments were conducted to analyze the performance of decomposition scheduling approach. The results show that in many cases decomposition scheduling performs better than the traditional branch-and-bound algorithms in terms of scheduling cost, and heuristic algorithms in terms of percentage of finding feasible schedules over randomly-generated task sets.

14. SUBJECT TERMS	15. NUMBER OF PAGES 17
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT Unlisted

Viewserver Hierarchy:
A New Inter-Domain Routing Protocol and its Evaluation*

Cengiz Alaettinoğlu† A. Udaya Shankar

Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
College Park, Maryland 20742

October 1993

Abstract

A simple approach to inter-domain routing is domain-level source routing with link-state approach where each node maintains a domain-level view of the internetwork. This does not scale up to large internetworks. The usual scaling technique of aggregating domains into superdomains loses ToS and policy detail.

We present a new viewserver hierarchy and associated protocols that (1) satisfies policy and ToS constraints, (2) adapts to dynamic topology changes including failures that partition domains, and (3) scales well to large number of domains without losing detail. Domain-level views are maintained by special nodes called viewservers. Each viewserver maintains a domain-level view of a surrounding precinct. Viewservers are organized hierarchically. To obtain domain-level source routes, the views of one or more viewservers are merged (upto a maximum of twice the levels in the hierarchy).

We also present a model for evaluating inter-domain routing protocols, and apply this model to compare our viewserver hierarchy against the simple approach. Our results indicate that the viewserver hierarchy finds many short valid paths and reduces the amount of memory requirement by two orders of magnitude.

Categories and Subject Descriptors: C.2.1 [Computer-Communication Networks]: Network Architecture and Design—*packet networks; store and forward networks*; C.2.2 [Computer-Communication Networks]: Network Protocols—*protocol architecture*; C.2.m [Routing Protocols]; F.2.m [Computer Network Routing Protocols].

1 Introduction

A computer internetwork, such as the Internet, is an interconnection of backbone networks, regional networks, metropolitan area networks, and stub networks (campus networks, office networks and other small networks)[1]. Stub networks are the producers and consumers of the internetwork traffic, while backbones, regionals, and MANs are transit networks. (Most of the networks in an internetwork are stub networks.) Each network consists of nodes (hosts, routers) and links. Two networks are *neighbors* when there is one or more links between nodes in the two networks (see Figure 1).
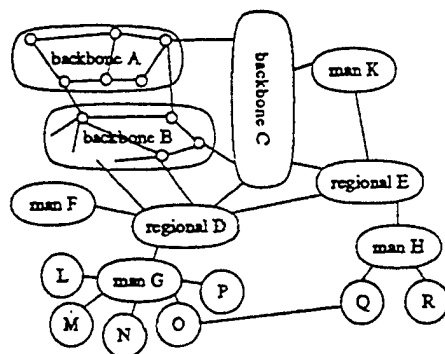

Figure 1: A portion of an internetwork. (Circles represent stub networks.)

An internetwork is organized into *domains*[2]. A domain is a set of networks (possibly consisting of only one network) administered by the same agency. Within each domain, an *intra-domain routing protocol* is executed that provides routes between source and destination nodes in the domain. This protocol can be any of the typical ones, i.e., next-hop or source routes computed using distance-vector or link-state algorithms.

Across all domains, an *inter-domain routing protocol* is executed that provides routes between source and destination nodes in different domains. This protocol must satisfy various constraints:

(1) It must satisfy *policy constraints*, which are administrative restrictions on the inter-domain traffic [8, 12, 9, 5]. Policy constraints are of two types: *transit policies* and *source policies*. The transit policies of a domain A specify how other domains can use the resources of A (e.g. $0.01 per packet, no traffic from domain B). The source policies of a domain A specify

---

[1] For example, NSFNET, MILNET are backbones and Suranet, CerfNet are regionals.
[2] also referred to as *routing domains*

constraints on traffic originating from A (e.g. domains to avoid/prefer, acceptable connection cost). Transit policies of a domain are public (i.e. available to other domains), whereas source policies are usually private.

(2) An inter-domain routing protocol must also satisfy *type-of-service* (ToS) constraints of applications (e.g. low delay, high throughput, high reliability, minimum monetary cost). To do this, it must keep track of the types of services offered by each domain [5].

(3) Inter-domain routing protocols must scale up to very large internetworks, i.e. with a very large number of domains. Practically this means that processing, memory and communication requirements should be much less than linear in the number of domains.

(4) Inter-domain routing protocols must automatically adapt to link cost changes, node/link failures and repairs including failures that partition domains [15]. It must also handle non-hierarchical domain interconnections at any level [9] (e.g. we do not want to hand-configure special routes as "back-doors").

A *simple* (or straightforward) approach to inter-domain routing is domain-level source routing with link-state approach [8, 5]. In this approach, each router[3] maintains a *domain-level view* of the internetwork, i.e., a graph with a vertex for every domain and an edge between every two neighbor domains. Policy and ToS information is attached to the vertices and the edges of the view.

When a source node needs to reach a destination node, it (or a router[4] in the source's domain) first examines this view and determines a *domain-level source route* satisfying ToS and policy constraints, i.e., a sequence of domain ids starting from the source's domain and ending with the destination's domain. Then, the packets are routed to the destination using this domain-level source route and the intra-domain routing protocols of the domains crossed.

The disadvantage of this simple scheme is that it does not scale up for large internetworks. The storage at each router is proportional to $N_D \times E_D$, where N_D is the number of domains and E_D is the average number of neighbor domains to a domain. The communication cost is proportional to $N_R \times E_R$, where N_R is the number of routers in the internetwork and E_R is the average router neighbors of a router (topology changes are flooded to all routers in the internetwork).

To achieve scaling, several approaches based on aggregating domains into superdomains have

---

[3] Not all nodes maintain routing tables. A router is a node that maintains a routing table.
[4] referred to as the policy server in [8]

been proposed [13, 16, 6]. This approaches have drawbacks because the aggregation results in loss of detail (discussed in Section 2).

Our protocol

In this paper, we present an inter-domain routing protocol that we have proposed recently[3]. It combines domain-level views with a novel hierarchical scheme. It scales well to large internetworks, and does not suffer from the problems of superdomains.

In our scheme, domain-level views are not maintained by every router but by special nodes called *viewservers*. For each viewserver, there is a subset of domains around it, referred to as the viewserver's *precinct*. The viewserver maintains the domain-level view of its precinct. This solves the scaling problem for storage requirement.

A viewserver can provide domain-level source routes between source and destination nodes in its precinct. Obtaining a domain-level source route between a source and a destination that are not in any single view, involves accumulating the views of a sequence of viewservers. To make this process efficient, viewservers are organized hierarchically in levels, and an associated addressing structure is used. Each node has a set of addresses. Each *address* is a sequence of viewserver ids of decreasing levels, starting at the top level and going towards the node. The idea is that when the views of the viewservers in an address are merged, the merged view contains domain-level routes to the node from the top level viewservers. (Addresses are obtained from name servers in the same way as is currently done in the Internet.)

We handle dynamic topology changes such as node/link failures and repairs, link cost changes, and domain partitions. Gateways[5] detect domain-level topology changes affecting its domain and neighbor domains. For each domain, there is a *reporting gateway* that communicates these changes by flooding to the viewservers in a specified subset of domains; this subset is referred to as its *flood area*. Hence, the number of packets used during flooding is proportional to the size of the flood area. This solves the scaling problem for the communication requirement.

Thus our inter-domain routing protocol consists of two subprotocols: a **view-query** protocol between routers and viewservers for obtaining merged views; and a **view-update** protocol between gateways and viewservers for updating domain-level views.

---

[5] A node is called a gateway if it has a link to another domain.

Evaluation

Many inter-domain routing protocols have been proposed, based on various kinds of hierarchies. How do these protocols compare against each other and against the simple approach? To answer this question, we need a model in which we can define internetwork topologies, policy/ToS constraints, inter-domain routing hierarchies, and evaluation measures (e.g. memory and time requirements) for inter-domain routing protocols. None of these protocols have been evaluated in a way that they can be compared against each other or the simple approach.

In this paper, we present such a model, and use it to compare our viewserver hierarchy to the simple approach. Our evaluation measures are the amount of memory required at the source and at the routers, the amount of time needed to construct a path, and the number of valid paths found (and their lengths) in comparison to the number of available valid paths (and their lengths) in the internetwork. We use three internetwork topologies each of size 11,110 domains (roughly the current size of the Internet). Our results indicate that the viewserver hierarchy finds many short valid paths and reduces the amount of memory requirement by two orders of magnitude.

Organization of the paper

In Section 2, we survey recent approaches to inter-domain routing. In Section 3, we present the view-query protocol for static network conditions, that is, assuming all links and nodes of the network remain operational. In Section 4, we present the view-update protocol to handle topology changes (this section is not needed for the evaluation part). In Section 5, we present our evaluation model and results from its application to the viewserver hierarchy. In Section 6, we conclude and describe how to add fault-tolerance and cacheing schemes to improve performance.

2 Related Work

In this section, we survey recently proposed inter-domain routing protocols that support ToS and Policy routing for large internetworks [14, 16, 13, 10, 6, 20, 2, 19, 18, 7].

Several inter-domain routing protocols (e.g. BGP [14], IDRP [16], NR [10]) are based on *path-vector* approach [17]. Here, for each destination domain a router maintains a set of paths, one through each of its neighbor routers. ToS and policy information is attached to these paths. Each

router requires $O(N_D \times N_D \times E_R)$ space. For each destination, a router exchanges its best valid path[6] with its neighbor routers. However, a path-vector algorithm may not find a valid path from a source to the destination even if such a route exists [13][7]. By exchanging k paths to each destination, the probability of detecting a valid path for each source can be increased.

The most common approach to solve the scaling problem is to use *superdomains*[8] (e.g. IDPR [13], IDRP [16], Nimrod [6]). Superdomains extend the idea of *area hierarchy* [11]. Here, domains are grouped hierarchically into superdomains: "close" domains are grouped into level 1 superdomains, "close" level 1 superdomains are grouped into level 2 superdomains, and so on. Each domain A is addressed by concatenating the superdomain ids starting from a top level superdomain and going down towards A. A router maintains a view that contains the domains in the same level 1 superdomain, the level 1 superdomains in the same level 2 superdomain, and so on. Thus a router maintains a smaller view than it would in the absence of hierarchy. Each superdomain has its own ToS and policy constraints derived from that of the subdomains.

There are several major problems with using superdomains. One problem is that if there are domains with different (possibly contradictory) constraints in a superdomain, then there is no good way of deriving the ToS and policy constraints of the superdomain. The usual techniques are to take either the union or the intersection of the constraints of the subdomains [13]. Both techniques have problems[9]. Other problems are described in [6, 2]. Some of the problems can be relaxed by having overlapping superdomains, but this increases the storage requirements drastically.

Nimrod [6] and IDPR [13] use the link-state approach, domain-level source routing, and super-domains (non-overlapping superdomains for Nimrod). IDRP [16] uses path-vector approach and superdomains.

Reference [10] combines the benefits of path-vector approach and link-state approach by having two modes: An NR mode, which is an extension of IDRP and is used for the most common ToS and policy constraints; and a SDR mode, which is like IDPR and is used for less frequent ToS and

---

[6] A valid path is a path that satisfies the ToS and policy constraints of the domains in the path.

[7] For example, suppose a router u has two paths $P1$ and $P2$ to the destination. Let u have a router neighbor v, which is in another domain. u chooses and informs v of one of the paths, say $P1$. But $P1$ may violate source policies of v's domain, and $P2$ may be a valid path for v.

[8] also referred to as routing domain confederations

[9] For example, if the union is taken, then a subdomain A can be forced to obey constraints of other subdomains; this may eliminate a path through A which is otherwise valid. If the intersection is taken, then a subdomain A can be forced to accept traffic it would otherwise not accept.

policy requests. This study does not address the scalability of the SDR mode.

In [2], we proposed another protocol based on superdomains. It always finds a valid path if one exists. Both union and intersection policy and ToS constraints are maintained for each visible superdomain. If the union policy constraints of superdomains on a path are satisfied, then the path is valid. If the intersection policy constraints of a superdomain are satisfied but the union policy constraints are not, the source uses a query protocol to obtain a more detailed "internal" view of the superdomain, and searches again for a valid path. The protocol uses a link-state view update protocol to handle topology changes, including failures that partition superdomains at any level.

The landmark hierarchy [19, 18] is another approach for solving the scaling problem. Here, each router is a landmark with a radius, and routers which are within a radius away from the landmark maintain a route to it. Landmarks are organized hierarchically, such that the radius of a landmark increases with its level, and the radii of top level landmarks include all routers. Addressing and packet forwarding schemes are introduced. Link-state algorithms can not be used with the landmark hierarchy, and a thorough study of enforcing ToS and policy constraints with this hierarchy has not been done.

The landmark hierarchy may look similar to our viewserver hierarchy, but in fact they are quite opposite. In the landmark hierarchy, nodes within the radius of the landmark maintain a route to the landmark, and the landmark may not have a route to these nodes. In the viewserver hierarchy, viewserver maintains routes (i.e. a view) to the nodes in its precinct.

Route fragments [7] is an addressing scheme. A destination route fragment, called a *route suffix*, is a sequence of domain ids from a backbone to the destination domain. A source route fragment, called a *route prefix*, is the reverse of a route suffix of that domain. There are also *route middles*, which are from transit domains to transit domains. These addresses are static (i.e. they are not updated with topology changes) and stored at the name servers. A source queries a name server and obtains destination route suffixes. It then chooses an appropriate route suffix for the destination and concatenates it with its own route prefix (and uses routes middles if route suffix and route prefix do not intersect). This scheme can not handle topology changes and does not address handling policy and ToS constraints.

3 Viewserver Hierarchy Query Protocol

In this section, we present our scheme for static network conditions, that is, all links and nodes remain operational. The dynamic case is presented in Section 4.

Conventions: Each domain has a unique id. DomainIds denotes the set of domain-ids. Each node has an id which is unique in its domain. NodeIds denotes the set of node-ids. Thus, a node is *totally* identified by the combination of its domain's id and its node-id. TotalIds denotes the set of total node-ids. For a node u, we use $domainid(u)$ to denote the domain-id of u's domain. We use $nodeid(u)$ and $totalid(u)$ to denote the node-id and total-id of u respectively. For a domain A, we use $domainid(A)$ to denote the domain-id of A. $NodeNeighbors(u)$ denotes the set of node-ids of the neighbors of u. $DomainNeighbors(A)$ denotes the set of domain-ids of the domain neighbors of A. We use the term gateway-id (or viewserver-id) to mean the total-id of a gateway node (or a viewserver node).

In our protocol, a node u uses two kinds of sends. The first kind has the form "Send(m) to v", where m is the message being sent and v is the total-id of the destination. Here, nodes u and v are neighbors, and the message is sent over the physical link (u, v). If the link is down, we assume that the packet is dropped.

The second kind of send has the form "Send(m) to v using $dlsr$", where m and v are as above and $dlsr$ is a domain-level source route between u and v. Here, the message is sent using the intra-domain routing protocols of the domains in $dlsr$ to reach v[10]. We assume that as long as there is a sequence of up links connecting the domains in $dlsr$, the message is delivered to v[11]. If the u and v are in the same domain, $dlsr$ equals $\langle\rangle$.

Views and Viewservers

Domain-level views are maintained by special nodes called *viewservers*. Each viewserver has a *precinct*, which is a set of domains around the viewserver, and a *static view*, which is a domain-level view of the precinct and outgoing edges. The static view includes the ToS and policy constraints

---

[10] Recall that given a domain-level source route to a destination, using the intra-domain routing protocols we can reach the destination.

[11] This involves time-outs, retransmissions, etc. It requires a transport protocol support such as TCP.

of domains in the precinct and of domain-level edges[12]. Formally, a viewserver x maintains the following:

$Precinct_x$. ($\subseteq$ DomainIds). Domain-ids whose view is maintained.

$SView_x$. Static view of x.
$$= \{\langle A, policy\&tos(A), \{\langle B, edge\_policy\&tos(A,B)\rangle : B \in subset\ of\ DomainNeighbors(A)\}\rangle :$$
$$A \in Precinct_x\}$$

$SView_x$ can be implemented as adjacency list representation of graphs [1]. The intention of $SView_x$ is to obtain domain-level source routes between nodes in $Precinct_x$. Hence, the choice of domains to include in $Precinct_x$ and the choice of neighbors of domains to include in $SView_x$ is not arbitrary. $Precinct_x$ and $SView_x$ must be connected; that is, between any two domains in $Precinct_x$, there should be a path in $SView_x$ that lies in $Precinct_x$. Note that $SView_x$ can contain edges to domains outside $Precinct_x$. We say that a domain A is *in the view* of a viewserver x, if either A is in the precinct of x or $SView_x$ has an edge from a domain in precinct to A. Note that the precincts and views of different view servers can be overlapping, identical or disjoint.

If there is a viewserver x whose view contains both the source and the destination domains, then x's view can be used to obtain the required domain-level source route to reach the destination. The source needs to reach x to obtain its view. If the source and x are in the same domain, x can be reached using the intra-domain routing protocol. If x is in another domain, then the source needs to have a domain-level source route to it[13]. In this case, we assume that source has a set of fixed domain-level source routes to x.

Viewserver Hierarchy

For scaling reasons, we cannot have one large view. Thus, obtaining a domain-level source route between a source and a destination which are far away, involves accumulating views of a sequence of viewservers. To keep this process efficient, we organize viewservers hierarchically. More precisely, each viewserver is assigned a hierarchy level from $0, 1, \ldots$, with 0 being the top level in the hierarchy. A parent/child relationship between viewservers is defined as follows:

---

[12] Not all the domain-level edges need to be included. This is because some domains may have many neighbors causing a big storage requirement.

[13] We cannot obtain this domain-level source route from x, i.e. chicken-egg problem.

1. Every level i viewserver, $i > 0$, has a parent viewserver whose level is less than i.

2. If viewserver x is a parent of viewserver y then x's view contains y's domain and y's view contains x's domain[14].

3. The view of a top level viewserver contains the domains of all other top level viewservers. (typically, top level viewservers are placed in backbones).

Note that the second constraint does not mean that all top level viewservers have the same view. In the hierarchy, a parent can have many children and a child can have many parents. We extend the range of the parent-child relationship to ordinary nodes; that is if the $Precinct_x$ contains the domain of node u, we say that u is a child of x, and x is a parent of u (note that an ordinary node does not have a child). We assume that there is at least one parent viewserver for each node.

For a node u, an address is defined to be a sequence $(x_0, x_1, \ldots, x_t)$ such that x_i for $i < t$ is a viewserver-id, x_0 is a top level viewserver-id, x_t is the total-id of u, and x_i is a parent of x_{i+1}. Note that a node may have many addresses since the parent-child relationship is many-to-many. If a source wants a domain-level source route to a destination, it first queries the name servers [15] to obtain a set of addresses for the destination. Then, it queries viewservers to obtain an accumulated view containing both its domain and the destination's domain.

Querying the name servers can be done the same way it is done currently in the Internet. It requires nodes to have a set of fixed addresses to name servers. This is also sufficient in our case. However, we can improve the performance by having a set of fixed domain-level source routes instead.

View-Query Protocol: Obtaining Domain-Level Source Routes

We now describe how a domain-level source route is obtained (regardless of whether the source and the destination are in a common view or not).

We want a sequence of viewservers whose merged views contains both the source and the destination domains. Addresses provide a way to obtain such a sequence, by first going up in the viewserver hierarchy starting from the source node and then going down in the viewserver hierarchy towards the destination node. More precisely, let $\langle s_0, \ldots, s_t \rangle$ be an address of the source,

---

[14] Note that x and y do not have to be in each other's precinct.

[15] In fact, name servers are called *domain name servers*. However, domain names and the domains used in this paper are different. We use "name servers" to avoid confusion.

and $\langle d_0, \ldots, d_l \rangle$ be an address of the destination. Then, the sequence $\langle s_{t-1}, \ldots, s_0, d_0, \ldots, d_{l-1} \rangle$ meets our requirements.[16] In fact, going up all the way in the hierarchy to top level viewservers may not be necessary. We can stop going up at a viewserver s_i if there is a viewserver $d_j, j < l$ such that the domain of d_j is in the view of s_i (one special case is where $s_i = d_j$).

The view-query protocol uses two message types:

- (RequestView, *s_address*, *d_address*)

  where *s_address* and *d_address* are the addresses for the source and the destination respectively. A RequestView message is sent by a source to obtain an accumulated view containing both the source and the destination domains. When a viewserver receives a RequestView message, it either sends back its view or forwards this request to another viewserver.

- (ReplyView, *s_address*, *d_address*, *accumview*)

  where *s_address* and *d_address* are as above and *accumview* is the accumulated view. A ReplyView message is sent by a viewserver to the source or to another viewserver closer to the source. The *accumview* field in a ReplyView message equals the union of the views of the viewservers the message has visited.

We now describe the events of a source node (see Figure 2). The source node[17] sends a RequestView packet containing a source and a destination address to its parent in the source address (using a fixed domain-level source route). When the source receives a ReplyView packet, it chooses a valid path using the *accumview* in the packet. If it does not find a valid path, it can try again using a different source and/or destination address. Note that, the source does not have to throw away the previous accumulated views, but merge all accumulated views into a richer accumulated view. In fact, it is easy to change the protocol so that source can also obtain views of individual viewservers to make the accumulated view even richer.

The events of a viewserver x are specified in Figure 3. Upon receiving a RequestView packet, x checks if the destination domain is in its precinct[18]. If it is, x sends back its view in a ReplyView packet. If it is not, x forwards the request packet to another viewserver as follows: x checks if the domain of any viewserver in the destination address is in its view or not. If there is such a domain,

---

[16] This is similiar to matching route fragments[7]. However, in our case the sequence is computed in a distributed fashion (these is needed to handle topology changes).

[17] or the policy server in the source's domain

[18] Even though destination can be in the view of x, its policies and ToS's are not in the view if it is not in the precinct of x.

291

Constants

$FixedRoutes_u(x)$, for every viewserver-id x such that x is a parent of u,

$$= \begin{cases} \{\langle\rangle\} & \text{if } domainid(u) = domainid(x) \\ \{\langle d_1, \ldots, d_n \rangle : d_i \in \text{DomainIds}\}. \text{ Set of domain-level routes to } x & \text{otherwise} \end{cases}$$

Events

$RequestView_u(s\_address, d\_address)$ {Executed when u wants a valid domain-level source route}
     Let $s\_address$ be $\langle s_0, \ldots, s_{t-1}, s_t \rangle$, and $dlsr \in FixedRoutes_u(s_{t-1})$;
     Send(RequestView, $s\_address$, $d\_address$) to s_{t-1} using $dlsr$

$Receive_u$(ReplyView, $s\_address$, $d\_address$, $accumview$)
     Choose a valid domain-level source route using $accumview$;
     If a valid route is not found
         Execute $RequestView_u$ again with another source address and/or destination address

Figure 2: View-query protocol: Events and state of a source u.

Constants

$Precinct_x$. Precinct of x.

$SView_x$. Static view of x.

Events

$Receive_x$(RequestView, $s\_address$, $d\_address$)
     Let $d\_address$ be $\langle d_0, \ldots, d_t \rangle$;
     if $domainid(d_t) \notin Precinct_x$ then
         $forward_x$(RequestView, $s\_address$, $d\_address$, $\{\}$);
     else $forward_x$(ReplyView, $d\_address$, $s\_address$, $SView_x$);   {addresses are switched}
     endif

$Receive_x$(ReplyView, $s\_address$, $d\_address$, $view$)
     $forward_x$(ReplyView, $s\_address$, $d\_address$, $view \cup SView_x$)

where procedure $forward_x$(type, $s\_address$, $d\_address$, $view$)
     Let $s\_address$ be $\langle s_0, \ldots, s_t \rangle$, $d\_address$ be $\langle d_0, \ldots, d_l \rangle$;
     if $\exists i : domainid(d_i)$ in $SView_x$ then
         Let $i = \max\{j : domainid(d_j)$ in $SView_x\}$;
         $target := d_i$;
     else $target := s_i$ such that $s_{i+1} = totalid(x)$;
     endif;
     $dlsr :=$ choose a route to $domainid(target)$ from $domainid(x)$ using $SView_x$;
     if $type = $ RequestView then
         Send(RequestView, $s\_address$, $d\_address$) to $target$ using $dlsr$;
     else Send(ReplyView, $s\_address$, $d\_address$, $view$) to $target$ using $dlsr$;
     endif

Figure 3: View-query protocol: Events and state of a viewserver x.

x sends the RequestView packet to the last such one. Otherwise x is a viewserver in the source

address and sends the packet to its parent in the source address. (Note that if x is a viewserver in the destination address, its child in the destination address is definitely in its view.)

When a viewserver x receives a ReplyView packet, it merges its view to the accumulated view in the packet. Then it sends the ReplyView packet towards the source node same way it would send a RequestView packet towards the destination node (i.e. the role of the source address and the destination address are changed).

Above we have described one possible way of obtaining the accumulated views. There are various other possibilities, for example: (1) restricting the ReplyView packet to take the reverse of the path that the RequestView packet took; (2) having ReplyView packets go all the way up in the viewserver-hierarchy for a richer accumulated view; (3) source polling the viewservers directly instead of viewservers forwarding request/reply messages to each other; (4) not including the non-transit stub domains other than the source and the destination domains in the *accumview*; (5) including some source policy constraints and ToS requirements in the RequestView packet, and having the viewservers filter out some domains.

4 Update Protocol for Dynamic Network Conditions

In this section, we first examine how topology changes such as link/node failures, repairs, and cost changes, map into domain-level topology changes. Second, we describe how domain-level topology changes are detected and communicated to viewservers, i.e. view-update protocol. Third, we modify the view-query protocol appropriately.

Mapping Topology Changes to Domain-Level Topology Changes

Costs are associated with domain-level edges. The cost of the domain-level edge (A, B) equals a vector of values if the link is up; each cost value indicates how expensive it is to cross domain A to reach domain B according to some criteria such as delay, throughput, reliability, etc. The cost equals ∞ if all links from A to B are down[19]. Each cost value of a domain-level edge (A, B) can be derived from the cost values of the intra-domain routes in A and links from A to B [4][20].

---

[19] Note that if a gateway connecting A to B is down, its links are also considered to be down.

[20] For example, the delay of a domain-level edge (A, B) can be calculated as the maximum/average delay of the routes from any gateway in A to first gateway in B.

Link cost changes and link/node failures and repairs correspond to cost changes, failures and repairs of domain-level edges. Link/node failures can also partition a domain into cells[15]. A *cell* is a maximal subset of nodes of a domain that can reach each other without leaving the domain. With partitioning, some nodes as well as some neighbor domains may not be accessible by all cells. In the same way, link/node repairs may merge cells into bigger cells. We identify a cell with the minimum node-id of the gateways in the cell. [21] In this paper, for uniformity we treat an unpartitioned domain as a domain with one cell; we do not consider cells that do not isolate gateways since such cells do not affect inter-domain routes.

If a domain gets partitioned, its vertex in the domain-level views should be split into as many pieces as there are cells. And when the cells merge, the corresponding vertices should be merged as well.

Since a domain can be partitioned into many cells, domain-level source routes now include cell-ids as well. Hence, the intra-domain routing protocol of a domain should include a route to each reachable neighbor domain cell.[22]

View-Update Protocol: Updating Domain-Level Views

Viewservers do not communicate with each other to maintain their views. Gateways detect and communicate domain-level topology changes to viewservers. Each gateway periodically (and optionally after a change in the intra-domain routing table) inspects its intra-domain routing table and determines the cell it belongs. For each cell, only the gateway whose node-id is the cell-id (i.e. the gateway with the minimum node-id) is responsible for communicating domain-level topology changes. We refer to this gateway as the *reporting gateway*. Reporting gateways compute the domain-level edge costs for each neighbor domain cell, and report them to parent viewservers. They are also responsible for informing the viewservers of the creation and deletion of cells.

The communication between a reporting gateway and viewservers is done by flooding over a set of domains. This set is referred to as the flood area[23]. The topology of a flood area must

---

[21] Our cells are like the domain components of IDPR[13].

[22] This involves the following changes in the intra-domain routing protocol: (1) Whenever the cell-id of a gateway changes, it reports its new cell-id to adjacent gateways in neighbor domains. When they receive this information, they update their intra-domain routes to include the new cell-id. (2) Usually when a node recovers from a failure, it queries its neighbors in its domain for their intra-domain routes. When a gateway recovers, it should also query adjacent gateways in neighbor domains for their cell-ids.

[23] For efficiency, the flood area can be implemented by a radius and some forwarding limits (e.g. do not flood

be a connected graph. Due to the nature of flooding, a viewserver can receive information out of order for a domain cell. In order to avoid old information replacing new information, each gateway includes successively increasing time stamps in the messages it sends.

Due to node and link failures, communication between a reporting gateway and a viewserver can fail, resulting in the viewserver having out-of-date information. To eliminate such information, a viewserver deletes any information about a domain cell if it is older than a *time-to-die* period. We assume that gateways send messages more often than the time-to-die value (to avoid false removal).

When a viewserver learns of a new domain cell, it adds it to its view. To avoid adding a domain cell which was just deleted[24], when a viewserver receives a delete domain cell request, it only marks the domain cell as deleted (and removes the entry after the time-to-die period).

The view-update protocol uses two types of messages as follows:

- (UpdateCell, *domainid, cellid, timestamp, floodarea, ncostset*)

  is sent by the reporting gateway to inform the viewservers about current domain-level edge costs of its cell. Here, *domainid, cellid,* and *timestamp* indicate the domain, the cell and the time stamp of the reporting gateway, *ncostset* contains a cost for each neighbor domain cell, and *floodarea* is the set of domains that this message is to be sent over.

- (DeleteCell, *domainid, cellid, timestamp, floodarea*)

  where the parameters are as in the UpdateCell message. It is sent by a reporting gateway when it becomes non-reporting (because its cell expanded to include a gateway with lower id), to inform viewservers to delete the gateway's old cell.

The state maintained by a gateway g is listed in Figure 4. Note that $LocalViewservers_g$ and $LocalGateways_g$ can be empty. $IntraDomainRT_g$ contains a route (next-hop or source) for every reachable node of the domain and for every reachable neighbor domain cell[25]. We assume that consecutive reads of $Clock_g$ returns increasing values.

The state maintained by a viewserver x is listed in Figure 5. $DView_x$ is the dynamic part of x's view. For each domain cell[26] known to x, $DView_x$ stores a *timestamp* field which equals the

---

beyond backbones) instead of a set.

[24] If the domain cell was removed, the timestamp for that domain cell is also lost.

[25] $IntraDomainRT_g$ is a view in case of a link-state routing protocol or a distance table in case of a distance-vector routing protocol.

[26] We use $A:g$ to denote the cell g of domain A.

```
Constants:
    LocalViewservers_g. (⊆ TotalIds). Set of viewservers in g's domain.
    LocalGateways_g. (⊆ TotalIds). Set of gateways in g's domain excluding g.
    AdjForeignGateways_g. (⊆ TotalIds). Set of adjacent gateways in other domains.
    FloodArea_g. (⊆ DomainIds). The flood area of the domain (includes domain of g).
Variables:
    IntraDomainRT_g. Intra-domain routing table of g. Initially contains no entries.
    CellId_g : NodeIds. The id of g's cell. Initially = ∞
    Clock_g : Integer. Clock of g.
```

Figure 4: State of a gateway g.

```
Constants:
    Precinct_x. Precinct of x.
    SView_x. Static view of x.
    TimeToDie_x : Integer. Time-to-die value.
Variables:
    DView_x. Dynamic view of x.
        = {⟨A:g, timestamp, expirytime, deleted,
              {⟨B:h, cost⟩ : B ∈ DomainNeighbors(A) ∧ h ∈ NodeIds ∪ {*} }⟩ :
              A ∈ Precinct_x ∧ g ∈ NodeIds}
    Clock_x : Integer. Clock of x.
```

Figure 5: State of a viewserver x.

largest timestamp received for this domain cell, an *expirytime* field which equals the end of the time-to-die period for this domain cell, a *deleted* field which marks whether or not the domain cell is deleted, and a cost set which indicates a cost for every neighbor domain cell whose domain is in $SView_x$. The cell-id of a neighbor domain equals * if no cell of the neighbor domain is reachable.

The events of gateway g and a viewserver x are specified in Appendix A.

Changes to View-Query Protocol

We now enumerate the changes needed to adapt the view-query protocol to the dynamic case (the formal specification is omitted for space reasons).

Due to link and node failures, RequestView and ReplyView packets can get lost. Hence, the

source may never receive a ReplyView packet after it initiates a request. Thus, the source should try again after a time-out period.

When a viewserver receives a RequestView message, in the static case it replies with its view if the destination domain is in its precinct. Now, because domain-level edges can fail, it must also check its dynamic view and reply with its views only if its dynamic view contains a path to the destination. Similarly during forwarding of RequestView and ReplyView packets, a viewserver, while checking whether a domain is in its view, should also check if its dynamic view contains a path to it.

Finally, when a viewserver sends a message to a node whose domain is partitioned, it should send a copy of the message to each cell of the domain. This is because a viewserver does not know which cell contains the node.

5 Evaluation

Many inter-domain routing protocols have been proposed, based on various kinds of hierarchies. How do these protocols compare against each other and against the simple approach? To answer this question, we need a model in which we can define internetwork topologies, policy/ToS constraints, inter-domain routing hierarchies, and evaluation measures (e.g. memory and time requirements) for inter-domain routing protocols.

In this section, we first present such a model, and then use the model to evaluate our viewserver hierarchy and compare it to the simple approach. Our evaluation measures are the amount of memory required at the source and at the routers, the amount of time needed to construct a path, and the number of paths found out of the total number of valid paths.

Even though the model described here can be applied to other inter-domain routing protocols, we have not done so, and hence have not compared them against our viewserver hierarchy. This is because of lack of time, and because precise definitions of the hierarchies in these protocols is not available. For example, to do a fair evaluation of IDPR[13], we need precise guidelines for how to group domains into super-domains, and how to choose between the union and intersection methods when defining policy/ToS constraints of super-domains. In fact, these protocols have not been evaluated in a way that we can compare them to the viewserver hierarchy. To the best of our knowledge, this paper is the first to evaluate a hierarchical inter-domain routing protocol against

explicitly stated policy constraints.

5.1 Evaluation Model

We first describe our method of generating topologies and policy/ToS constraints. We then describe the evaluation measures.

Generating Internetwork Topologies

For our purposes, an internetwork *topology* is a directed graph where the nodes correspond to domains and the edges correspond to domain-level connections. However, an arbitrary graph will not do. The topology should have the characteristics of a real internetwork, like the Internet. That is, it should have backbones, regionals, MANS, LANS, etc.; these should be connected hierarchically (e.g. regionals to backbones), but "non-hierarchical" connections (e.g. "back-doors") should also be present.

For brevity, we refer to backbones as class 0 domains, regionals as class 1 domains, metropolitan-area domains and providers as class 2 domains, and campus and local-area domains as class 3 domains. A (strictly) hierarchical interconnection of domains means that class 0 domains are connected to each other, and for $i > 0$, class i domains are connected to class $i - 1$ domains. As mentioned above, we also want some "non-hierarchical" connections, i.e., domain-level edges between domains irrespective of their classes (e.g. from a campus domain to another campus domain or to a backbone domain).

In reality, domains span geographical regions and domain-level edges are usually between domains that are geographically close (e.g. University of Maryland campus domain is connected to SURANET regional domain which is in the east cost). A class i domain usually spans a larger geographical region than a class $i + 1$ domain. To generate such interconnections, we associate a "region" attribute to each domain. The intention is that two domains with the same region are geographically close.

The *region* of a class i domain has the form $r_0.r_1.\cdots.r_i$, where the r_j's are integers. For example, the region of a class 3 domain can be 1.2.3.4. For brevity, we refer to the region of a class i domain as a class i region.

Note that regions have their own hierarchy. Class 0 regions are the top level regions. We say

that a class i region $r_0.r_1.\cdots.r_i$ is *contained* in the class $i-1$ region $r_0.r_1.\cdots.r_{i-1}$ (where $i > 0$). Containment is transitive. Thus region 1.2.3.4 is contained in regions 1.2.3, 1.2 and 1.
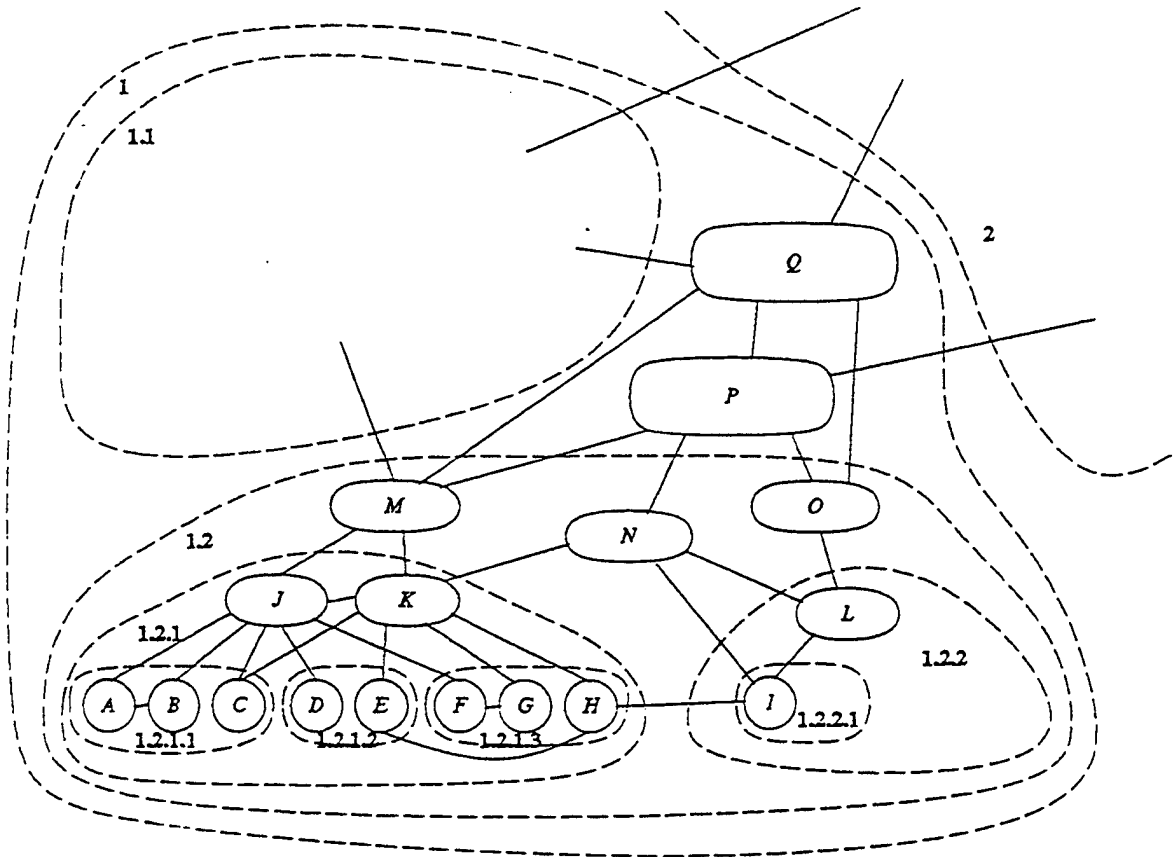

Figure 6: Regions

Given any pair of domains, we classify them as local, remote or far, based on their regions. Let X be a class i domain and Y a class j domain, and (without loss of generality) let $i \leq j$. X and Y are *local* if they are in the same class i region. For example in Figure 6, A is local to B, C, J, K, M, N, O, P, and Q. X and Y are *remote* if they are not in the same class i region but they are in the same class $i - 1$ region, or if $i = 0$. For example in Figure 6, some of the domains A is remote to are D, E, F, and L. X and Y are *far* if they are not local or remote. For example in Figure 6, A is far to I.

We refer to a domain-level edge as *local* (*remote*, or *far*) if the two domains it connects are local

(remote, or far).

We use the following procedure to generate internetwork topologies:

- We first specify the number of domain classes, and the number of domains in each class.

- We next specify the regions. Note that the number of region classes equals the number of domain classes. We specify the number of class 0 regions. For each class $i > 0$, we specify a *branching factor*, which creates that many class i regions in each class $i - 1$ region. (That is, if there are two class 0 regions and the class 1 branching factor equals three, then there are six class 1 regions.)

- For each class i, we randomly map the class i domains into the class i regions. Note that several domains can be mapped to the same region, and some regions may have no domain mapped into them.

- For every class i and every class j, $j \geq i$, we specify the number of local, remote and far edges to be introduced between class i domains and class j domains. The end points of the edges are chosen randomly (within the specified constraints).

We ensure that the internetwork topology is connected by ensuring that the subgraph of class 0 domains is connected, and each class i domain, for $i > 0$, is connected to a local class $i - 1$ domain.

Choosing Policy/ToS Constraints

We chose a simple scheme to model Policy/ToS constraints. Each domain is assigned a color: *green* or *red*. For each domain class, we specify the percentage of green domains in that class, and then randomly choose a color for each domain in that class.

A *valid route* from a source to a destination is one that does not visit any red intermediate domains; the source and destination are allowed to be red. Notice that this models transit policy/ToS constraints. We are working on extending this model to source policy/ToS constraints.

Computing Evaluation Measures

The evaluation measures of most interest for an inter-domain routing protocol are its memory and time requirements, and the number of valid paths it finds (and their lengths) in comparison to the number of available valid paths (and their lengths) in the internetwork (e.g. could it find the

shortest valid path in the internetwork).

The only analysis method we have at present is to numerically compute the evaluation measures for a variety of source-destination pairs. Because we use internetwork topologies of large sizes, it is not feasible to compute for all possible source-destination pairs. We randomly choose a set of source-destination pairs that satisfy the following conditions: (1) the source and destination domains are different, and (2) there exists a valid path from the source domain to the destination domain in the internetwork topology. (Note that the simple scheme would always find such a path.)

For a source-destination pair, we refer to the length of the shortest valid path in the internetwork topology as the *shortest-path length*. Since the number of paths between a source-destination pair is potentially very large (factorial in the number of domains), and we are not interested in the paths that are too long, we only count the number of paths whose lengths are not more than the shortest-path-length plus 2.

The evaluation measures described above are protocol independent. However, there are also important evaluation measures that are protocol dependent (e.g. number of levels traversed in some particular hierarchy). Because of this we postpone the precise definitions of the evaluation measures to the next subsection (their definition is dependent of viewserver hierarchy).

5.2 Application to Viewserver Protocol

We have used the above model to evaluate our viewserver protocol for several different viewserver hierarchies and query methods. We first describe the different viewserver schemes evaluated. Please refer to Figure 6 in the following discussion.

The first viewserver scheme is referred to as base. It has exactly one viewserver in each domain. Each viewserver is identified by its domain-id. The domains in a viewserver's precinct consist of its domain and the neighboring domains. The edges in the viewserver's view consist of the edges between the domains in the precinct, and edges outgoing from domains in the precinct to domains not in the precinct. For example, the precinct of viewserver A (i.e. the viewserver in domain A) consists of domains A, B, J; the edges in the view of viewserver A consists of domain-level edges $(A, B), (A, J), (B, J), (J, M), (J, K), (J, F)$, and (J, D).

As for the viewserver hierarchy, a viewserver's level is defined to be the class of its domain. That is, a viewserver in a class i domain is a level i viewserver. For each level i viewserver, $i > 0$, its

parent viewserver is chosen randomly from the level $i - 1$ viewservers in the parent region such that there is a domain-level edge between the viewserver's domain and the parent viewserver's domain. For example, for viewserver C, we can pick viewserver J or K; suppose we pick J. For viewserver J, we have no choice but to pick M (N and O are not connected to J). For M, we pick P (out of P and Q).

We use only one address for each domain. The viewserver-address of a stub domain is concatenation of four viewserver (i.e. domain) ids. Thus, the address of A is $P.M.J.A$. Similarly, the address of H is $P.M.K.H$. To obtain a route between A and H, it suffices to obtain views of viewservers A, J, K, H.

The second viewserver scheme is referred to as **base-QT** (where the QT stands for "query upto top"). It is identical to *base* except that during the query protocol all the viewservers in the source and the destination addresses are queried. That is, to obtain a route between A and H, the views of A, J, M, P, K, H are obtained.

The third viewserver scheme is referred to as **locals**. It is identical to *base* except that now a viewserver's precinct also contains domains that have the same region as the viewserver's domain. That is, the precinct of viewserver A has the domains A, B, J, C. Note that in this scheme a viewserver's view is not necessarily connected. For example, if the edge (C, J) is removed, the view of viewserver A is no longer connected. (In Section 3, we said that the view of a viewserver should be connected. Here we have relaxed this condition to simplify testing.)

The fourth viewserver scheme is referred to as **locals-QT**. It is identical to *locals* except that during the query protocol all the viewservers in the source and the destination addresses are queried.

The fifth viewserver scheme is referred to as **vertex-extension**. It is identical to *base* except that viewserver precincts are extended as follows: Let P denote the precinct of a viewserver in the *base* scheme. For each domain X in P, if there is an edge from domain X to domain Y and Y is not in P, domain Y is added to the precinct; among Y's edges, only the ones to domains in P are added to the view. In the example, domains M, K, F, D are added to the precinct of A, but outgoing edges of these domains to other domains are not included (e.g. (F, G) is not included). The advantage of this scheme is that even though it increases the precinct size by a factor which is potentially greater than 2, it increases the number of edges stored in the view by a factor less than 2. (In fact, if the same edge cost and edge policies are used for both directions of domain-

level edges, then the only other information that needs to be stored by the viewservers is the policy constraints of the newly added domains.)

The sixth viewserver scheme is referred to as full-QT. It is constructed in the same way as *vertex-extension* except that the *locals* scheme is used instead of *base* scheme to define the P in the construction. In full-QT, during the query protocol all the viewservers in the source and the destination addresses are queried.

In all the above viewserver schemes, we have used the same hierarchy for both domain classes and viewservers. In practice, not all domains need to have a viewserver, and a viewserver hierarchy different from the domain class hierarchy can be deployed. However, there is an advantage of having a viewserver in each domain; that is, source nodes do not require fixed domain-level source routes to their parent viewservers (in the view-query protocol). This reduces the amount of hand configuration required. In fact, the *base* scheme does not require any hand configuration, viewservers can decide their precincts from the intra-domain routing tables, and nodes can use intra-domain routes to reach parent viewservers.

Results for Internetwork 1

The parameters of the first internetwork topology, referred to as Internetwork 1, are shown in Table 1.

Our evaluation measures were computed for a (randomly chosen but fixed) set of 1000 source-destination pairs. For brevity, we use *spl* to refer to the *shortest-path length* (i.e. the length of the shortest valid path in the internetwork topology). The minimum *spl* of these pairs was 2, the maximum *spl* was 13, and the average *spl* was 6.8. Table 2 lists for each viewserver scheme (1) the minimum, average and maximum precinct sizes, (2) the minimum, average and maximum merged view sizes, and (3) the minimum, average and maximum number of viewservers queried.

The precinct size indicates the memory requirement at a viewserver. More precisely, the memory requirement at a viewserver is $O(\text{precinct size} \times d)$ where d is the average number of neighbor domains of a domain, except for the *vertex-extension* and *full-QT* schemes. In these schemes, the memory requirement is increased by a factor less than two. Hence the *vertex-extension* scheme has the same order of viewserver memory requirement as the *base* scheme and the *full-QT* scheme has

---

[27] Branching factor is 4 for all region classes.

Class i	No. of Domains	No. of Regions[27]	% of Green Domains	Edges between Classes i and j			
				Class j	Local	Remote	Far
0	10	4	0.80	0	8	6	0
1	100	16	0.75	0	190	20	0
				1	26	5	0
2	1000	64	0.70	0	100	0	0
				1	1060	40	0
				2	200	40	0
3	10000	256	0.20	0	100	0	0
				1	100	0	0
				2	10100	50	0
				3	50	50	50

Table 1: Parameters of Internetwork 1.

Scheme	Precinct Size	Merged View Size	No. of Viewservers Queried
base	2 / 3.2 / 68	7 / 71.03 / 101	3 / 7.51 / 8
base-QT	2 / 3.2 / 68	30 / 76.01 / 101	8 / 8.00 / 8
locals	2 / 52.0 / 103	3 / 95.40 / 143	2 / 7.42 / 8
locals-QT	2 / 52.0 / 103	43 / 101.86 / 143	8 / 8.00 / 8
vertex-extension	3 / 19.2 / 796	23 / 362.15 / 486	3 / 7.51 / 8
full-QT	11 / 102.9 / 796	228 / 396.80 / 519	8 / 8.00 / 8

Table 2: Precinct sizes, merged view sizes, and number of viewservers queried for Internetwork 1.

the same order of viewserver memory requirement as the *locals* scheme.

The merged view size indicates the memory requirement at a source; i.e. the memory requirement at a source is O(merged view size $\times d$) except for the *vertex-extension* and *full-QT* schemes. Note that the source does not need to store information about red and non-transit domains. The numbers in Table 2 take advantage of this.

The number of viewservers queried indicates the communication time required to obtain the merged view at the source. Because the average *spl* is 6.8, the "real-time" communication time

required to obtain the merged view at a source is slightly more than one round-trip time between the source and the destination.

As is apparent from Table 2, using a QT scheme increases the merged view size and the number of viewservers queried only by about 5%. Using a *locals* scheme increases the merged view size by about 30%. Using the *vertex-extension* scheme increases the merged view size by 5 times (note that the amount of actual memory needed increases only by a factor less than 2). The number of viewservers queried in the *locals* scheme is less than the number of viewservers queried in the *base* scheme. This is because the viewservers in the *locals* scheme have bigger precincts, and a path from the source to the destination can be found using fewer views.

Table 3 shows the average number of *spl*, *spl* + 1, *spl* + 2 length paths found for a source-destination pair by the simple approach and by the viewserver schemes. All the viewserver schemes are very close to the simple approach. The *vertex-extension* and *full-QT* schemes are especially close (they found 98% of all paths). Table 3 also shows the number of pairs for which the viewserver schemes did not find a path (ranging from 1.4% to 5.9% of the source-destination pairs), and the number of pairs for which the viewserver schemes found longer paths. For these pairs, more viewserver addresses need to be tried. Note that the *locals* and *vertex-extension* schemes decrease the number of these pairs substantially (adding QT yields further improvement). Our policy constraints are source and destination domain independent. Hence, even a class 2 domain, if it is red, can not carry traffic to a class 3 domain to which it is connected. We believe that these figures would improve with policies that are dependent on source and destination domains.

As is apparent from Table 3 and Table 2, the *locals* scheme does not find many more extra paths than the *base* scheme even though it has larger precinct and merged view sizes. Hence it is not recommended. The *vertex-extension* scheme is the best, but even *base* is adequate since it finds many paths.

We have repeated the above evaluations for two other internetworks and obtained similar conclusions. The results are in Appendix B.

6   Concluding Remarks

We presented hierarchical inter-domain routing protocol that (1) satisfies policy and ToS constraints, (2) adapts to dynamic topology changes including failures that partition domains, and

Scheme	Number of paths found			No. of pairs with no path	No. of pairs with longer paths
	spl	$spl+1$	$spl+2$		
simple	2.51	18.48	131.01	N/A	N/A
base	2.41	15.84	99.42	59	3 by 1.33 hops
base-QT	2.41	15.86	100.16	54	3 by 1.33 hops
locals	2.41	16.17	103.54	29	3 by 1 hop
locals-QT	2.41	16.29	105.02	20	3 by 1 hop
vertex-extension	2.51	18.38	128.19	22	0 by 0 hops
full-QT	2.50	18.40	128.90	14	0 by 0 hops

Table 3: Number of paths found for Internetwork 1.

(3) scales well to large number of domains.

Our protocol uses partial domain-level views to achieve scaling in space requirement. It floods domain-level topological changes over a flood area to achieve scaling in communication requirement.

It does not abstract domains into superdomains; hence it does not lose any domain-level detail in ToS and policy information. It merges a sequence of partial views to obtain domain-level source routes between nodes which are far away. The number of views that need to be merged is bounded by twice the number of levels in the hierarchy.

To evaluate and compare inter-domain routing protocols against each other and against simple approach, we presented a model in which one can define internetwork topologies, policy/ToS constraints, inter-domain routing hierarchies, and evaluation measures. We applied this model to evaluate our viewserver hierarchy and compared it to the simple approach. Our results indicate that viewserver hierarchy finds many short valid paths and reduces the amount of memory requirement by two order of magnitude.

Our protocol recovers from fail-stop failures of viewservers and gateways. When a viewserver fails, an address which includes the viewserver's id becomes useless. This deficiency can be overcome by replicating each viewserver at different nodes of the domain (in this case a viewserver fails only if all nodes implementing it fail). This replication scheme requires viewserver ids to be independent of node ids, which can be easily accomplished[28].

---

[28] For example, if node-ids of nodes implementing a viewserver share a prefix, this prefix can be used as the

The only drawback of our protocol is that to obtain a domain-level source route, views are merged at (or prior to) the connection (or flow) setup, thereby increasing the setup time. This drawback is not unique to our scheme [8, 13, 6, 10].

There are several ways to reduce the setup overhead. First, domain-level source routes to frequently used destinations can be cached. The cacheing period would depend on the ToS requirement of the applications and the frequency of domain-level topology changes. For example, the period can be long for electronic mail since it does not require shortest paths.

Second, views of frequently queried viewservers can be replicated at "mirror" viewservers in the source domain. A viewserver would periodically update the views of its mirror viewservers.

Third, connection setup also involves traversing the name server hierarchy (to obtain destination addresses from its names). By integrating the name server hierarchy with the viewserver hierarchy, we may be able to do both operations simultaneously. This requires further investigation.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ulman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] C. Alaettinoğlu and A. U. Shankar. Hierarchical Inter-Domain Routing Protocol with On-Demand ToS and Poicy Resolution. In Proc. *IEEE International Conference on Networking Protocols '93*, San Fransisco, California, October 1993.

[3] C. Alaettinoğlu and A. U. Shankar. Viewserver Hierarchy: A Scalable and Adaptive Inter-Domain Routing Protocol. Technical Report UMIACS-TR-93-13, CS-TR-3033, Department of Computer Science, University of Maryland, College Park, February 1993.

[4] A. Bar-Noy and M. Gopal. Topology Distribution Cost vs. Efficient Routing in Large Networks. In Proc. *ACM SIGCOMM '90*, pages 242–252, Philadelphia, Pennsylvania, September 1990.

[5] L. Breslau and D. Estrin. Design of Inter–Administrative Domain Routing Protocols. In Proc. *ACM SIGCOMM '90*, pages 231–241, Philadelphia, Pennsylvania, September 1990.

[6] J. N. Chiappa. A New IP Routing and Addressing Architecture. Big-Internet mailing list., 1992. Available by anonymous ftp from munnari.oz.an:big-internet/list-archive.

[7] D. Clark. Route Fragments, A Routing Proposal. Big-Internet mailing list., July 1992. Available by anonymous ftp from munnari.oz.au:big-internet/list-archive.

[8] D.D. Clark. Policy routing in Internet protocols. Request for Comment RFC-1102, Network Information Center, May 1989.

[9] D. Estrin. Policy requirements for inter Administrative Domain routing. Request for Comment RFC-1125, Network Information Center, November 1989.

[10] D. Estrin, Y. Rekhter, and S. Hotz. Scalable Inter-Domain Routing Architecture. In Proc. *ACM SIGCOMM '92*, pages 40–52, Baltimore, Maryland, August 1992.

viewserver id. Intra-domain routing would forward a packet destined to a viewserver to any operational node with this prefix.

[11] F. Kamoun and L. Kleinrock. Stochastic Performance Evaluation of Hierarchical Routing for Large Networks. *Computer Networks and ISDN Systems*, 1979.

[12] B.M. Leiner. Policy issues in interconnecting networks. Request for Comment RFC-1124, Network Information Center, September 1989.

[13] M. Lepp and M. Steenstrup. An Architecture for Inter-Domain Policy Routing. Internet Draft. Available from the authors., June 1992.

[14] K. Lougheed and Y. Rekhter. Border Gateway Protocol (BGP). Request for Comment RFC-1105, Network Information Center, June 1989.

[15] R. Perlman. Hierarchical Networks and Subnetwork Partition Problem. *Computer Networks and ISDN Systems*, 9:297–303, 1985.

[16] Y. Rekhter. Inter-Domain Routing Protocol (IDRP). Available from the author., 1992. T.J. Watson Research Center, IBM Corp.

[17] K. G. Shin and M. Chen. Performance Analysis of Distributed Routing Strategies Free or Ping-Pong-Type Looping. *IEEE Transactions on Computers*, 1987.

[18] P. F. Tsuchiya. The Landmark Hierarchy: Description and Analysis, The Landmark Routing: Architecture Algorithms and Issues. Technical Report MTR-87W00152, MTR-87W00174, The MITRE Corporation, McLean, Virginia, 1987.

[19] P. F. Tsuchiya. The Landmark Hierarchy:A New Hierarchy For Routing In Very Large Networks. In Proc. *ACM SIGCOMM '88*, August 1988.

[20] P. F. Tsuchiya. Efficient and Robust Policy Routing Using Multiple Hierarchical Addresses. In Proc. *ACM SIGCOMM '91*, pages 53–65, Zurich, Switzerland, September 1991.

A View-Update Protocol Event Specifications

The events of gateway g are specified in Figure 7. When a gateway g recovers, $CellId_g$ is set to $nodeid(g)$. Thus, when g next executes $Update_g$, it sends either an UpdateCell or a DeleteCell message to viewservers, depending on whether it is no longer the minimum id gateway in its cell[29].

The events of a viewserver x are specified in Figure 5. Note that when x adds an entry to $DView_x$ (upon receiving a UpdateCell message), it selectively chooses subset of neighbors from the cost set in the packet to include only the neighbor domains which are in $SView_x$. When a viewserver x recovers, $DView_x$ is set to {}. Its view becomes up-to-date as it receives new information from reporting gateways (and remove false information with the time-to-die period).

---

[29] Sending a DeleteCell message is essential. Because prior to the failure, g may have been the smallest id gateway in its cell. Hence, some viewserver's may still contain an entry for its old domain cell.

```
Update_g        {Executed periodically and also optionally upon a change in IntraDomainRT_g}
    {Determines the id of g's cell and initiates UpdateCell and DeleteCell messages if needed.}
    OldCellId = CellId_g;
    CellId_g := compute cell id using LocalGateways_g and IntraDomainRT_g;
    if nodeid(g) = CellId_g then
        ncostset := compute costs for each neighbor domain cell using IntraDomainRT_g;
        flood_g((UpdateCell, domainid(g), CellId_g, Clock_g, FloodArea_g, ncostset));
    endif
    if nodeid(g) = OldCellId ≠ CellId_g then
        flood_g((DeleteCell, domainid(g), nodeid(g), Clock_g, FloodArea_g));
    endif

Receive_g(packet)        {either an UpdateCell or a DeleteCell packet}
    flood_g(packet)

where procedure flood_g(packet)
    if domainid(g) ∈ packet.floodarea then
        {remove domain of g from the flood area to avoid infinite exchange of the same message.}
        packet.floodarea := packet.floodarea — {domainid(g)};
        for all h ∈ LocalGateways_g ∪ LocalViewservers_g do
            Send(packet) to h using ();
    endif
    for all h ∈ AdjForeignGateways_g ∧ domainid(h) ∈ packet.floodarea do
        Send(packet) to h;

Gateway Failure Model:   A gateway can undergo failures and recoveries at anytime. We assume failures
are fail-stop (i.e. a failed gateway does not send erroneous messages). When a gateway g recovers, CellId_g
is set to nodeid(g).
```

Figure 7: View-update protocol: Events of a gateway g.

B   Results for Other Internetworks

Results for Internetwork 2

The parameters of the second internetwork topology, referred to as Internetwork 2, are the same as the parameters of Internetwork 1 (a different seed is used for the random number generation).

Our evaluation measures were computed for a set of 1000 source-destination pairs. The minimum spl of these pairs was 2, the maximum spl was 13, and the average spl was 7.2.

Table 4 and Table 5 shows the results. Similar conclusions to Internetwork 1 hold for Internetwork 2. In Table 5, the reason that local and QT schemes have more pairs with longer paths than the base scheme is that these schemes found some paths (which are not shortest) for some pairs for which the base scheme did not find any path.

```
Receive_x(UpdateCell, did, cid, ts, FloodArea, ncset)
    if did ∈ Precinct_x then
        if ∃⟨did:cid, timestamp, expirytime, deleted, ncostset⟩ ∈ DView_x ∧
            ts > timestamp then          {received is more recent;  delete the old one}
                delete ⟨did:cid, timestamp, expirytime, deleted, ncostset⟩ from DView_x;
        endif
        if ¬∃⟨did:cid, timestamp, expirytime, deleted, ncostset⟩ ∈ DView_x then
            Choose ncostset from ncset using SView_x;
                insert ⟨did:cid, ts, Clock_x + TimeToDie_x, false, ncostset⟩ to DView_x;
        endif
    endif

Receive_x(DeleteCell, did, cid, ts, floodarea)
    if did ∈ Precinct_x then
        if ∃⟨did:cid, timestamp, expirytime, deleted, ncostset⟩ ∈ DView_x ∧
            ts > timestamp then          {received is more recent;  delete the old one}
                delete ⟨did:cid, timestamp, expirytime, deleted, ncostset⟩ from DView_x;
        endif
        if ¬∃⟨did:cid, timestamp, expirytime, deleted, ncostset⟩ ∈ DView_x then
                insert ⟨did:cid, ts, Clock_x + TimeToDie_x, true, {}⟩ to DView_x;
        endif
    endif

Delete_x      {Executed periodically to delete entries older than the time-to-die period}
    for all ⟨A:g, tstamp, expirytime, deleted, ncset⟩ ∈ DView_x ∧ expirytime < Clock_x do
        delete ⟨A:g, tstamp, expirytime, deleted, ncset⟩ from DView_x;
```

Viewserver Failure Model: A viewserver can undergo failures and recoveries at anytime. We assume failures are fail-stop. When a viewserver x recovers, $DView_x$ is set to $\{\}$.

Figure 8: View update events of a viewserver x.

Scheme	Precinct Size	Merged View Size	No. of Viewservers Queried
base	2 / 3.2 / 76	4 / 66.62 / 96	3 / 7.55 / 8
base-QT	2 / 3.2 / 76	29 / 72.76 / 96	8 / 8.00 / 8
locals	3 / 69.8 / 149	4 / 101.32 / 148	2 / 7.36 / 8
locals-QT	3 / 69.8 / 149	35 / 110.32 / 152	8 / 8.00 / 8
vertex-extension	3 / 19.47 / 817	15 / 339.60 / 469	3 / 7.55 / 8
full-QT	11 / 135.2 / 817	186 / 402.51 / 503	8 / 8.00 / 8

Table 4: Precinct sizes, merged view sizes, and no of viewservers queried for Internetwork 2.

Results for Internetwork 3

The parameters of the third internetwork topology, referred to as Internetwork 3, are shown in Table 6. Internetwork 3 is more connected, more class 0, 1 and 2 domains are green, and more

Scheme	Number of paths found			No. of pairs with no path	No. of pairs with longer paths
	spl	$spl+1$	$spl+2$		
simple	2.21	13.22	74.30	N/A	N/A
base	1.98	8.20	34.40	123	13 by 1.08 hops
base-QT	1.98	8.36	35.62	110	15 by 1.13 hops
locals	2.08	9.18	40.50	97	23 by 1.39 hops
locals-QT	2.08	9.38	42.08	67	23 by 1.30 hops
vertex-extension	2.18	12.57	64.98	19	6 by 1 hop
full-QT	2.19	12.85	67.37	4	4 by 1 hop

Table 5: Number of paths found for Internetwork 2.

class 3 domains are red. Hence, we expect more valid paths between source and destination pairs.

Our evaluation measures were computed for a set of 1000 source-destination pairs. The minimum spl of these pairs was 2, the maximum spl was 10, and the average spl was 5.93.

Class i	No. of Domains	No. of Regions[30]	% of Green Domains	Edges between Classes i and j			
				Class j	Local	Remote	Far
0	10	4	0.85	0	8	7	0
1	100	16	0.80	0	190	20	0
				1	50	20	0
2	1000	64	0.75	0	500	50	0
				1	1200	100	0
				2	200	40	0
3	10000	256	0.10	0	300	50	0
				1	250	100	0
				2	10250	150	50
				3	200	150	100

Table 6: Parameters of Internetwork 3.

---

[30] Branching factor is 4 for all domain classes.

Table 7 and Table 8 shows the results. Similar conclusions to Internetwork 1 and 2 hold for Internetwork 3.

Scheme	Precinct Size	Merged View Size	No. of Viewservers Queried
base	2 / 3.5 / 171	5 / 134.41 / 206	3 / 7.26 / 8
base-QT	2 / 3.5 / 171	55 / 154.51 / 206	8 / 8.00 / 8
locals	3 / 70.17 / 171	4 / 164.16 / 257	2 / 7.09 / 8
locals-QT	3 / 70.17 / 171	57 / 191.06 / 258	8 / 8.00 / 8
vertex-extension	5 / 34.17 / 1986	18 / 601.56 / 695	3 / 7.26 / 8
full-QT	14 / 155.5 / 1986	503 / 655.79 / 743	8 / 8.00 / 8

Table 7: Precinct sizes, merged view sizes, and no of viewservers queried for Internetwork 3.

Scheme	Number of paths found			No. of pairs with no path	No. of pairs with longer paths
	spl	*spl* + 1	*spl* + 2		
simple	3.34	37.55	368.97	N/A	N/A
base	2.83	24.25	178.08	17	11 by 1.09 hops
base-QT	2.87	25.53	193.41	12	8 by 1.12 hops
locals	2.87	25.62	196.33	21	8 by 1 hop
locals-QT	2.97	27.59	219.63	2	6 by 1 hop
vertex-extension	3.32	35.73	332.54	5	1 by 1 hop
full-QT	3.33	36.47	346.44	0	0 by 0 hops

Table 8: Number of paths found for Internetwork 3.

Figure 9 through Figure 11 show the number of *spl*, *spl* + 1 and *spl* + 2 length paths found by the schemes as a function of *spl* (we only show results for *spl* values for which more than 10 pairs were found). We do not include *base-QT*, *locals* and *locals-QT* schemes since they are very close to *base* scheme. As expected, as *spl* increases, the number of paths for a source-destination pair increases, and the gap between the *simple* scheme and the viewserver schemes increases.
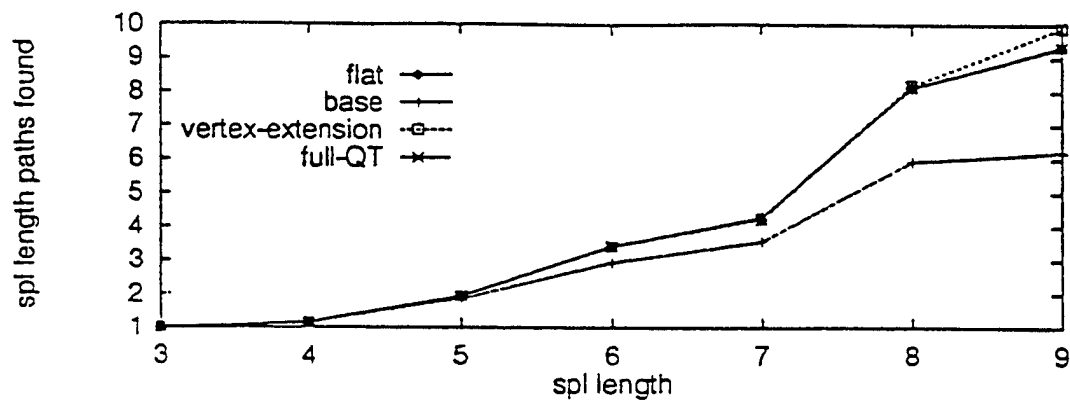
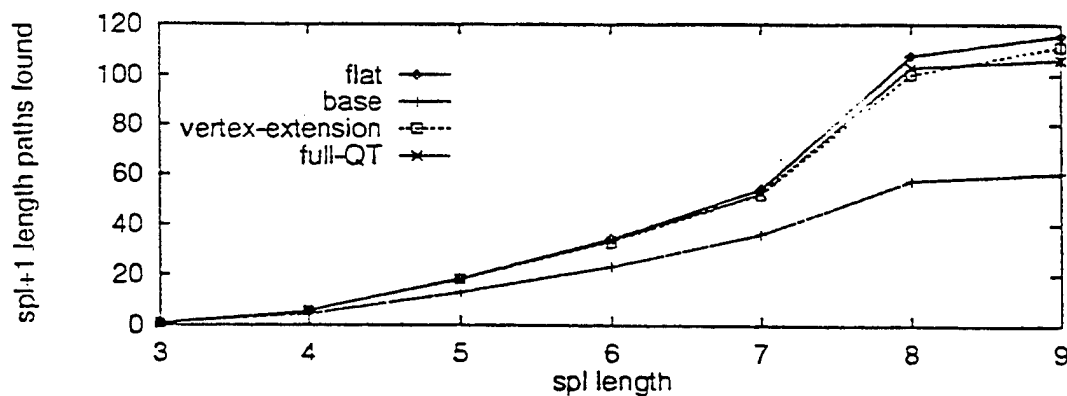Figure 9: Number of *spl* length paths found for Internetwork 3.

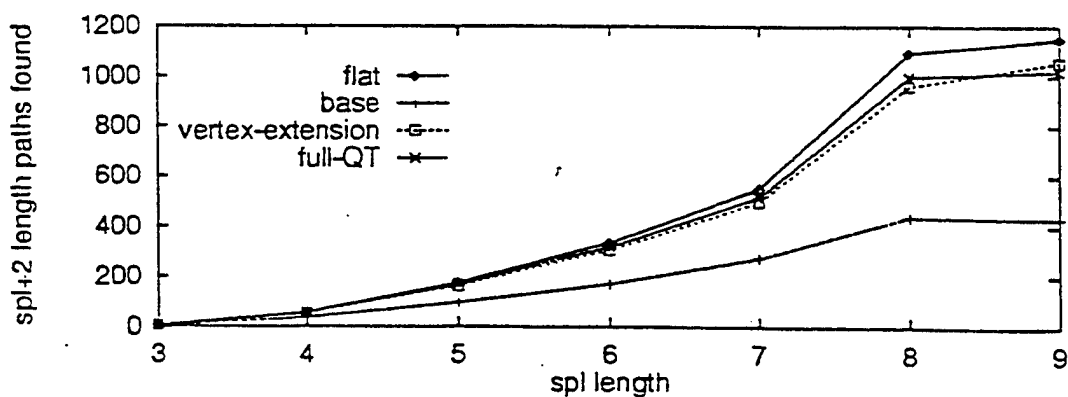Figure 10: Number of *spl* + 1 length paths found for Internetwork 3.

Figure 11: Number of *spl* + 2 length paths found for Internetwork 3.

REPORT DOCUMENTATION PAGE

Form approved OMB No 074-0188

1. AGENCY USE ONLY (leave blank)	2. REPORT DATE 10/15/93	3. REPORT TYPE END DATES COVERED Technical Report

4. TITLE AND SUBTITLE
Viewserver Hierarchy: A New Inter-Domain Routing Protocol and its Evaluation

5. FUNDING NUMNBERS
DASG-60-92-C-0055

6. AUTHOR(S)
Cengiz Alaettinoglu, and A. Udaya Shankar

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Maryland
Department of Computer Science
A.V. Williams Building
College Park, MD 20742

8. PERFORMING ORGANIZATION REPORT NUMBER
CS-TR-3151
UMIACS-TR-93-98

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)
PhillipsLaboratory
Director of Contracting
3651 Lowry Avenue SE
Kirtland AFB, NM 87117-5777

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12.a. DISTRIBUTION/ AVAILABILITY STATEMENT

12.b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

A simple approach to inter-domain routing is domain-level source routing with link-state approach where each node maintains a domain-level view of the internetwork. This does not scale up to large internetworks. The usual scaling tecnnique of aggregating domains into superdomains loses ToS and policy detail.

We present a new viewserver hierarchy and associated protocols that (1) satisfies policy and ToS constraints, (2) adapts to dynamic topology changes including failures that partition domains, and (3) scales well to large nuimber of dominas without losing detail. Domain-level views are maintained by special nodes called viewservers. Each viewserver maintains a domain-level view of surrounding precinct. Viewservers are organized hierarchically. To obtain domain-level source routes, the views of one or more viewservers are merged (upto a maximum of twice the levels in the hierarchy).

We also present a model for evaluating inter-domain routing protocols, and apply this model to compare our viewserver hierarchy against the simple approach . Our results indicate that the viewserver hierarchy finds many short valid paths and reduces the amount of memory requirement by two orders of magnitude.

14. SUBJECT TERMS
Network Architecture and Design

15. NUMBER OF PAGES
33

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT Unlimited

TEMPORAL ANALYSIS FOR HARD REAL-TIME SCHEDULING*

Manas C. Saksena and Ashok K. Agrawala

Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

Static time driven scheduling has been advocated for use in Hard Real-Time systems and is particularly appropriate for many embedded systems. The approaches taken for static scheduling often use search techniques and may reduce the search by using heuristics. In this paper we present a technique for analyzing the temporal relations among the tasks, based on non-preemptive schedulability. The relationships can be used effectively to reduce the average complexity of scheduling these tasks. They also serve as a basis for selective preemption policies for scheduling by providing an early test for infeasibility. We present examples and simulation results to confirm the usefulness of temporal analysis as a phase prior to scheduling.

1   Introduction

Many safety critical real-time applications like process control, embedded tactical systems for military applications, air-traffic control, robotics etc. have stringent timing constraints imposed on their computations due to the characteristics of the physical system. A failure to observe the timing constraints can result in intolerable system degradation and in some cases it may have catastrophic consequences.

Scheduling is the primary means of ensuring the satisfaction of timing constraints for such systems[1]. As a result, significant effort has been invested in research on hard real time scheduling [2, 3, 4]. In this paper we discuss a scheduling technique for static scheduling to guarantee timely execution of time critical tasks.

The time driven scheduling model is being used by many experimental systems, including MARS[5], MARUTI[6] and Spring Kernel[7]. The static time driven scheduling technique involves constructing a schedule offline, which may be represented as a Gantt chart[8] or calendar[6] (Figure 1). Tasks are invoked at run-time whenever they are scheduled to execute. Such a scheduling model is particularly appropriate for many embedded systems. Recent effort in this direction has shown the viability of such an approach for practical real-time applications[9].
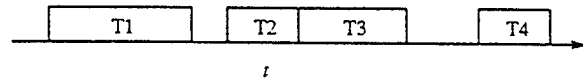
Figure 1: Gantt Chart or Calendar

The intractability of most scheduling problems has led to approaches based on search techniques for scheduling of real-time tasks. The feasibility of a task set is determined through construction of a schedule; failure to construct a schedule denotes infeasibility. Heuristics are often used as a means of controlling the complexity of scheduling. In many cases, heuristics perform well enough to result in an acceptable solution.

There has been little emphasis on the use of analytic techniques to assist in time driven scheduling. Decomposition scheduling[10] based on dominance properties of sequences[11] uses analytic techniques to decompose a set of tasks into a sequence of subsets. Significant reduction in average complexity can be achieved if the set of tasks can be decomposed into a large number of subsets, each having a small number of tasks.

In this paper, we present an analysis technique for time driven scheduling based on the timing requirements of tasks. The analysis results in the establishment of a set of temporal relations between pairs of tasks based on a non-preemptive scheduling model. These relations can be used by scheduling algorithms to reduce the complexity of scheduling in the average case, and as an early test for infeasibility. As a test for infeasibility, it provides a good basis for policies using selective preemption to enhance feasibility. When infeasibility is not detected, the temporal relations may be used by a search algorithm to effectively prune large portions of search space, thereby controlling the cost of scheduling.

2   Time Driven Scheduling

The time driven scheduling approach constructs a calendar for the set of tasks in the system. The tasks may be scheduled preemptively or non-preemptively. The non-preemptive scheduling problem for a uniprocessor is known to be NP-Complete[12]. When the tasks are mutually independent, and can be preempted at any time, it is known that the earliest deadline first policy is optimal[13] and obviates the need for non-preemptive scheduling. However, when tasks synchronize

315

using critical sections, the preemptive scheduling problem is also known to be intractable(NP-Hard)[14].

In general, when the overhead of preemption is negligible, the non-preemptive solutions form a subset of preemptive solutions[8]. However, when tasks may interact with each other, the non-preemptive models are simpler, easier to implement and closer to reality[15]. They are also necessary for certain scheduling domains like I/O scheduling and provide a basis for selective preemption policies.

2.1 Task Model

We consider a set of n tasks $\Gamma = \{\tau_i : i = 1, 2, \ldots, n\}$ to be scheduled for execution on a single processor. Each task τ_i, abbreviated as i, is a 3-tuple $[r_i, c_i, d_i]$ denoting the ready time, computation time and deadline respectively. The time interval $[r_i, d_i]$ is called the timing window w_i of task τ_i, and indicates the time interval during which the task can execute. The computation time of each task is less than the window length $|w_i|^1$. All tasks are assumed to be independent for simplicity of exposition even though such a requirement is not necessary for the analysis.

In a hard real-time system, processes may be *periodic* or *sporadic* [14]. Such a set of processes may be mapped to our scheduling model by techniques identified in [1, 14, 16] and constructing a schedule for the least common multiple of the periods of the tasks.

2.2 Non-Preemptive Scheduling Model

A non-preemptive schedule is the mapping of each task τ_i in Γ to a start time s_i. The task is then scheduled to run without preemption in the time interval $[s_i, f_i]$, with its finish time being $f_i = s_i + c_i$. A *feasible* schedule is a schedule in which the following conditions are satisfied for each task τ_i:

$$r_i \leq s_i \tag{1}$$
$$f_i \leq d_i \tag{2}$$

It is useful to consider a non-preemptive schedule as an ordered sequence of the set of tasks. To get a maximally packed schedule from a sequence $[\tau_1, \tau_2, \ldots, \tau_n]$, we can recursively derive the start time s_i and finish time f_i of the tasks as follows:

$$s_i = max(r_i, f_{i-1}) \tag{3}$$
$$f_i = s_i + c_i \tag{4}$$

with

$$s_1 = r_1$$

The scheduling problem can thus be considered as a search over the permutation space. A permutation (sequence) is feasible if the corresponding schedule is feasible. Notice that for any permutation schedule derived as above, equation 1 is implied by (3) and we only need to verify the deadline constraints for the tasks.
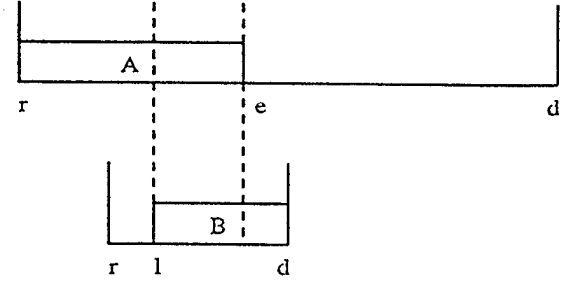

$^1|w_i| = d_i - r_i$

Figure 2: Infeasibility of task A executing before task B

3 Temporal Analysis

Temporal analysis uses pairwise schedulability analysis of tasks to generate a set of relations to eliminate sequences which cannot lead to feasible solutions. In this section we define the temporal relations and show how they may be derived from the timing constraints of tasks.

3.1 Definitions of Temporal Relations

Consider two tasks τ_i and τ_j. We wish to find out what we can say about the relative ordering of these tasks, given their timing constraints. A set of relations are identified below which identify the different possibilities.

Precedence Relation: A precedence relation denoted as $\tau_i \longrightarrow \tau_j$, implies that in any feasible schedule τ_i must execute before τ_j.

Infeasible Relation: An infeasible relation denoted by $\tau_i \oslash \tau_j$ implies that in any feasible schedule, τ_i and τ_j cannot run in a sequential order.

Concurrent Relation: $\tau_i \parallel \tau_j$ if there is no precedence or infeasible relation between them. A concurrent relation indicates that a feasible schedule may exist with any order of the tasks τ_i and τ_j. It does not, however, indicate the existence of a feasible schedule.

For each task τ_i let us define two terms e_i and l_i, denoting the *earliest finish time* and the *latest start time* as:

$$e_i = r_i + c_i \tag{5}$$
$$l_i = d_i - c_i \tag{6}$$

A preliminary set of relations can be established using the following rules, for every pair of tasks τ_i and τ_j.

$$(e_i \leq l_j) \wedge (l_i < e_j) \Rightarrow \tau_i \longrightarrow \tau_j \tag{7}$$
$$(e_i > l_j) \wedge (l_i \geq e_j) \Rightarrow \tau_j \longrightarrow \tau_i \tag{8}$$
$$(e_i \leq l_j) \wedge (l_i \geq e_j) \Rightarrow \tau_i \parallel \tau_j \tag{9}$$
$$(e_i > l_j) \wedge (l_i < e_j) \Rightarrow \tau_i \oslash \tau_j \tag{10}$$

The basic idea is that if the earliest finish time of a task A is greater than the latest start time of a task B, then a feasible schedule cannot be found in which A is scheduled before B
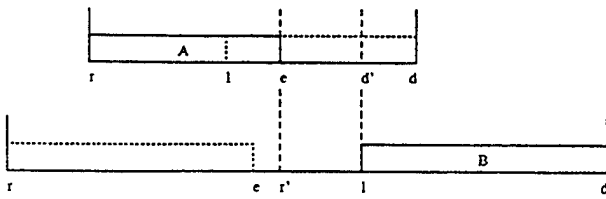
316

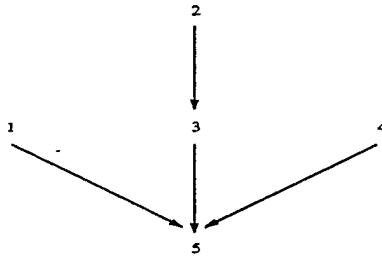Figure 3: Window Modification (A $\longrightarrow$ B)

Figure 5: Precedence Graph for example of Figure 4

(Figure 2). Thus, for instance the first part of condition for rule 10 says that τ_i cannot precede τ_j, and the second part says that τ_j cannot precede τ_i, establishing the infeasible relation.

3.2 Window Modification

Consider two tasks τ_a and τ_b, and a precedence relation $\tau_a \longrightarrow \tau_b$ between them. As this indicates that in any feasible schedule τ_a must precede τ_b, we can update the timing windows, as follows (Figure 3):

$$d'_a = min(d_a, l_b) \qquad (11)$$
$$r'_b = max(r_b, e_a) \qquad (12)$$

The window modification does not alter the scheduling problem in the sense that every feasible sequence with the original timing constraints is a feasible sequence with the modified timing constraints and vice-versa. Further, the schedules for feasible sequences are identical in both cases. A task's window may shrink because of window modification. This may lead to a change in the relation of the modified task with other tasks. The procedure may be applied iteratively till no further changes can be made or an infeasible relation is detected.

3.3 Examples

(a) Consider a set of five tasks as shown in Figure 4. The temporal analysis leads us to the following set of precedence relations, sans the redundant ones:

$$\{\tau_2 \longrightarrow \tau_3, \tau_1 \longrightarrow \tau_5, \tau_3 \longrightarrow \tau_5, \tau_4 \longrightarrow \tau_5\}$$

The set of precedence relations may be represented as a precedence graph (Figure 5) and impose a partial order on the task set. Only sequences which are consistent with this partial order need to be considered for scheduling. For 5 tasks, the total number of permutations is $120(= 5!)$. The number of total orders consistent with the partial order of Figure 5 is 12, which is a drastic reduction in the number of sequences that need to be considered for scheduling. The modified task set is shown in Figure 4(b), with the modified values in bold.

(b) As another example, consider the set of 4 tasks as shown in Figure 6. The task set in different stages of temporal analysis is shown, with the new temporal relations[2] at each stage. This example shows how successive refinement of temporal relations can lead to detecting infeasibility.

3.4 Complexity of Temporal Analysis

It is easy to see that the initial set of relations can be established in $O(n^2)$ time. Further, each phase of refinement also takes no more than $O(n^2)$. An upper bound for the number of phases is n. Therefore, the worst case complexity of temporal analysis is $O(n^3)$. In practice, however, the cost of temporal analysis can be significantly less since concurrent relations and relations between non-overlapping tasks need not be generated explicitly. Furthermore, the number of phases required to stabilize window modification can be reduced if the release times are modified in the topological sort order of the precedence graph and deadlines are modified in the reverse topological sort order.

In any case, the cost of temporal analysis for static scheduling is not significant when used in conjunction with an exponential time scheduling algorithm. In section 5, we show empirically that the cost of temporal analysis is not a significant factor for static scheduling.

4 Non Preemptive Scheduling using Temporal Analysis

The relations established through temporal analysis serve as a basis for scheduling of the tasks. Temporal analysis may thus be perceived as a pre-processing stage for scheduling. The result of this pre-processing stage is one of the following:

1. The task set was detected to be infeasible, due to the existence of one or more infeasible relations.

2. A set of precedence relations were established generating a precedence graph. The precedence graph imposes a partial order on the set of tasks. It serves as an input to the scheduler which may exploit the partial order generated to prune the search space.

4.1 Detecting Infeasibility

Whenever, an infeasible relation exists between two tasks, it is known that no ordering of the two tasks is feasible. Thus,

---

[2]Concurrent Relations are not shown.

317

	τ_1	τ_2	τ_3	τ_4	τ_5
r	0	5	9	0	8
c	7	8	4	8	13
d	29	16	23	30	42

(a)

	τ_1	τ_2	τ_3	τ_4	τ_5
r	0	5	13	0	17
c	7	8	4	8	13
d	29	16	23	29	42

(b)

Figure 4: Window Modification: (a) Original Task Set (b) Task Set after Temporal Analysis

	τ_1	τ_2	τ_3	τ_4
r	40	30	0	0
e	55	55	25	25
c	15	25	25	25
l	45	65	75	75
d	60	90	100	100
$\tau_1 \longrightarrow \tau_2$				

(a)

	τ_1	τ_2	τ_3	τ_4
r	40	55	0	0
e	55	80	25	25
c	15	25	25	25
l	45	65	75	75
d	60	90	100	100
$\tau_3 \longrightarrow \tau_2, \tau_4 \longrightarrow \tau_2$				

(b)

	τ_1	τ_2	τ_3	τ_4
r	40	30	0	0
e	55	55	25	25
c	15	25	25	25
l	45	65	40	40
d	60	90	65	65
$\tau_3 \longrightarrow \tau_1, \tau_4 \longrightarrow \tau_1$				

(c)

	τ_1	τ_2	τ_3	τ_4
r	40	55	0	0
e	55	80	25	25
c	15	25	25	25
l	45	65	20	20
d	60	90	45	45
$\tau_3 \oslash \tau_4$				

(d)

Figure 6: Example for Determining Infeasibility with Temporal Analysis

the detection of an infeasible relation at any stage in temporal analysis indicates that the task set is infeasible. Even though only pairwise schedulability analysis is used for establishing relations, successive refinement of relations results in a possible percolation of this effect to other tasks too. This effect is exemplified in the example of Figure 6, where several iterations lead to a infeasible relation. It must be noted that whenever infeasibility is detected, the resulting task set and their relations also provide a good feedback as to what caused it. The feedback information may be used to allocate more resources, change resource allocation or allow for selective preemption as the case may be.

4.2 Search Technique for Scheduling

The intractability of non-preemptive scheduling has led to implicit enumeration techniques based on branch and bound search methods. The search space is the set of all possible permutation sequences. One way of enumerating schedules is to generate an initial schedule and then successively refine it using heuristics to generate "better" schedules, until a feasible schedule is obtained [3, 17, 16].

In this paper, we concentrate on another enumeration method which constructs a schedule in an incremental manner. Variants of this method have been used in [4, 18, 19, 20, 21] The search space is represented as a search tree. The

root (level 0) of the tree is an empty schedule. The nodes of the tree represent partial schedules. A node at level k gives a partial schedule with k tasks. The leaves are complete schedules. The successors of an intermediate node are *immediate extensions* of the partial schedule corresponding to that node. From a node at level k, there are at most $n-k$ branches with each branch corresponding to an extension of the partial schedule by appending one more task to the schedule. Search is done in a *branch and bound* manner, wherein parts of the search tree are pruned when it is determined that no feasible schedule can arise from them. For each node being expanded, the following conditions must hold.

1. All immediate extensions of the node must be feasible [4, 18].

2. The remaining computational demand must not exceed the difference between the largest deadline of remaining tasks and current scheduling time [4].

If any condition is violated then no feasible schedule can be generated in the subtree originating from this node. No search is conducted on the subtree rooted at such a node.

4.2.1 Heuristically Guided Scheduling

Heuristics are commonly used to guide search in many combinatorial searching problems. For non-preemptive scheduling
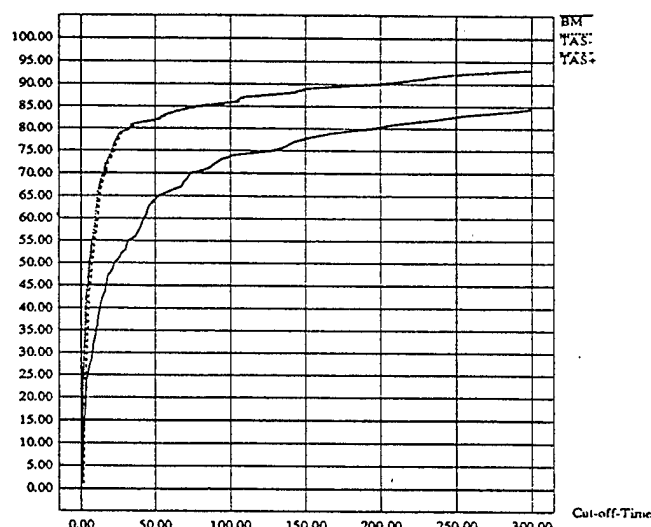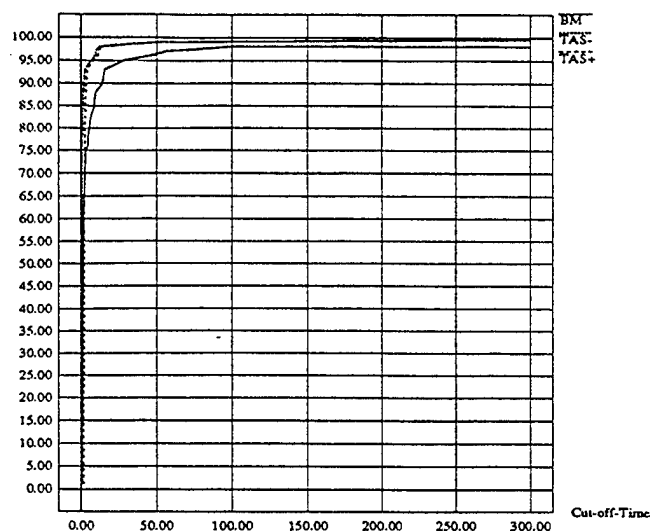
318

Figure 8: Success Ratio vs Cut-off-Time, $\mu_{\mathcal{L}} = 5.0\mu_{\mathcal{C}}$, $\mathcal{COV}_{\mathcal{L}} = 2.0$

Figure 9: Success Percentage vs Cut-off-Time, $\mu_{\mathcal{L}} = 10.0\mu_{\mathcal{C}}$, $\mathcal{COV}_{\mathcal{L}} = 1.0$

developing analytic techniques. Temporal analysis is a step in this direction and provides an efficient way of analyzing a task set and deducing valuable information for scheduling.

The existence of an infeasible relation in a task set gives a sufficient condition for infeasibility. This provides an early test for infeasibility, which can then be used as a basis for selective preemption to enhance feasibility. Alternatively, the detection of infeasibility may be used to allocate more resources or change resource allocation.

The precedence relations generated as a result of temporal analysis impose a partial order on the task set and may be effectively used to prune the search space for scheduling. Our simulations confirm that temporal analysis helps in improving the performance of a scheduling algorithm without incurring a significant overhead. In the simplest scheduling case, when heuristics perform very well, temporal analysis might be perceived as a way of formalizing the heuristics. For static time driven scheduling to be a feasible technique, it becomes imperative that the scheduling cost be controlled as the size of the problem increases. Temporal analysis provides a step in the right direction.

In this paper we have been concerned with single processor scheduling. An interesting extension of temporal analysis would be to use it for multi-processor scheduling. One way to extend the analysis to multi-processor scheduling is to perform it in two phases. In the first phase the infeasible and concurrent relations may be used to obtain an allocation of tasks to processors. Then in the second phase, the analysis shown in this paper can be used for each processor for scheduling.

Many real-time system specifications impose relative timing constraints on the tasks[23, 24]. In this paper, we have restricted ourselves to absolute constraints on the start and finish times of tasks. When more complex constraints are imposed on tasks, the role of temporal analysis in reducing the search space becomes even more important since simple heuristics are unlikely to perform well. It would be interesting to see how temporal analysis can be extended to use such constraints to further prune the search space.

We are currently implementing a scheduling tool based on the results shown in this paper. The tool is being developed for the MARUTI project, an experimental real-time system prototype being developed at the University of Maryland, based on the concept of pre-scheduling[6].

References

[1] J. Xu and D. L. Parnas, "On Satisfying Timing Constraints in Hard-Real-Time Systems", *in Proceedings of the ACM SIGSOFT'91 Conference on Software for Critical Systems*, pp. 132–146, December 1991.

[2] C. L. Liu and J. Layland, "Scheduling algorithm for multiprogramming in a hard real-time environment", *Journal of the ACM.*, vol. 20, pp. 46–61, Jan. 1973.

[3] J. Xu and D. L. Parnas, "Scheduling processes with release times, deadlines, precedence, and exclusion relations", *IEEE Transactions on Software Engineering*, vol. SE-16, pp. 360–369, March 1990.

[4] W. Zhao, K. Ramamritham, and J. A. Stankovic, "Scheduling Tasks with Resource requirements in a Hard Real-Time System", *IEEE Transactions on Software Engineering*, vol. SE-13, pp. 564–577, May 1987.

[5] H. Kopetz, A. Damm, Ch. Koza, M. Mulazzani, W. Schwabl, Ch. Senft, and R. Zainlinger, "Distributed

heuristics may be used to guide search along paths which are more likely to lead feasible schedules. Search is done in a depth first manner until either a complete feasible schedule is found, in which case the search terminates, or it is determined that no possible extensions of the current node can lead to a feasible schedule. Heuristics are used to determine which of the many children of a node should be searched next. Backtracking takes place when no further extensions of a node can be made. We evaluate temporal analysis using such a heuristic search for scheduling.

5 Empirical Evaluation of Temporal Analysis

In the previous sections, we have shown how temporal analysis may be used to restrict the search space for scheduling. Clearly, the existence of even a few precedence relations results in a drastic reduction of search space[3]. However, the usefulness of the scheme is not obvious since we are only interested in feasible schedules, hence a large part of the search space may never need to be examined. We have conducted various simulations to verify that indeed temporal analysis results in improved performance for scheduling. For reasons of space, we mention only a few significant results.

We used a heuristic search technique for scheduling as described in section 4.2. The heuristic used for our simulation study was a two level heuristic. The primary heuristic was *earliest start time*(EST).

$$EST_i = max(r_i, f_k)$$

where k is the last task in the partial schedule at that node.

In the case of a conflict, the secondary heuristic *earliest deadline* was used. Further conflicts were resolved arbitrarily. The heuristic has a natural intuitive appeal and is known to produce good results among linear heuristics[22].

For each set of parameters, we generated 200 "feasible" task sets with 100 tasks each. The task sets were generated with 100% utilization as this presents the most difficulty for scheduling. The computation times were generated using uniform distributions and laxities using normal distribution. We compared the *success percentage* (i.e. percentage of successfully scheduled task sets) of scheduling with and without temporal analysis as a pre-processing stage. The success percentage (SP) is plotted against "cut-off-time", indicating the maximum time allowed to the scheduling algorithm to successfully generate a schedule.

Our simulation results show that temporal analysis is not needed for scheduling when both the mean and the variation in laxities is low since the simple heuristics were able to schedule almost all task sets (success ratio ≈ 1.0). However, when the laxities are high (as compared to computation times) and the variation in laxities is also high[4], then the heuristics do
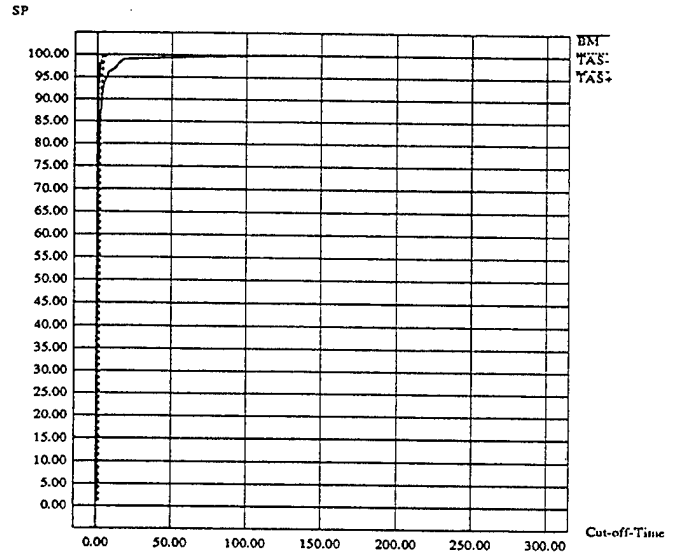
[3]Even one relation reduces the search space by half.
[4]Note that the task set utilization is 100%

Figure 7: Success Ratio vs Cut-off-Time, $\mu_{\mathcal{L}} = 5.0\mu_C$, $COV_{\mathcal{L}} = 1.0$

not perform as well and the use of temporal analysis results in $10 - 20\%$ improvement in success ratio.

As an illustration, we show a few plots which plot the success percentage (SP) of scheduling with temporal analysis (TAS) contrasted with success percentage of scheduling without temporal analysis, i.e. the baseline scheduling model (BM). For scheduling with temporal analysis, we consider two cases, one in which overhead of temporal analysis is added to scheduling time (TAS+) and the other in which it is not (TAS-). The parameters varied are the mean laxity $\mu_{\mathcal{L}}$ in terms of mean computation times μ_C, and the coefficient of variation for laxity $COV_{\mathcal{L}}$. Figures 7 and 8 show the plots for low laxity mean with low and high variation. For this case, there is no significant performance improvement due to temporal analysis and both schemes achieve almost 100% success percentage. On the other hand when the average laxity is high (Figures 9 and 10), coupled with high variation, we see that temporal analysis results in significant improvement in performance. The plots also show that the curves for (TAS+) and (TAS-) are almost identical showing that the overhead of temporal analysis is minimal when compared to the scheduling costs.

6 Concluding Remarks

In this paper we have presented *temporal analysis* as a technique for analyzing the timing relationships among a set of tasks to establish constraints on scheduling which are discernible from a pairwise analysis. The implications and the benefits of the approach as a pre-processing stage for scheduling has been shown through examples and simulation.

Time Driven Scheduling theory has relied heavily on search techniques for scheduling and little work has been done in
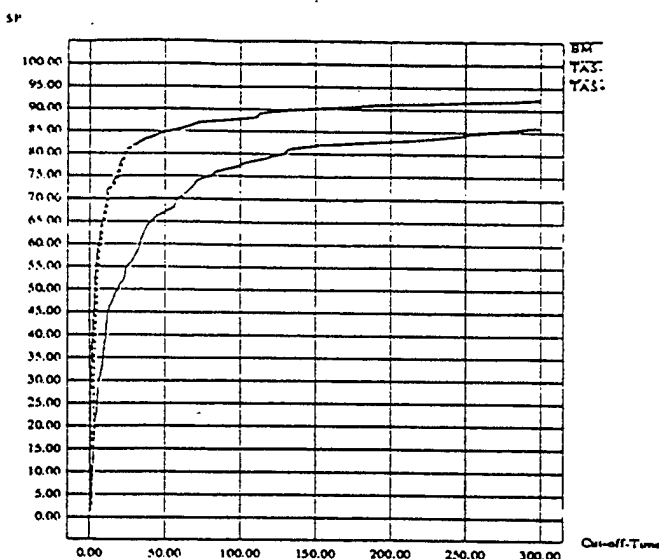
Figure 10: Success Percentage vs Cut-off-Time, $\mu_c = 10.0\mu_c$, $COV_c = 2.0$

Fault-Tolerant Real-Time Systems: The MARS Approach", *IEEE Micro*, vol. 9, pp. 25-40, Feb. 1989.

[6] S. T. Levi, S. K. Tripathi, S. D. Carson, and A. K. Agrawala, "The MARUTI Hard Real-Time Operating System", *ACM SIGOPS, Operating Systems Review*, vol. 23, pp. 90-106, July 1989.

[7] J. A. Stankovic and K. Ramamritham, "The Spring Kernel: A New Paradigm for Real-Time Operating Systems", *ACM SIGOPS, Operating Systems Review*, vol. 23, pp. 54-71, July 1989.

[8] E. G. Coffman, *Computer and Job-Shop Scheduling Theory*, Ed., Wiley, New York, 1976.

[9] T. Shepard and J. A. M. Gagne, "A Model of The F-18 Mission Computer Software for Pre-Run Time Scheduling", *in Proceedings IEEE 10th International Conference on Distributed Computer Systems*, pp. 62-69, May 1990.

[10] X. Yuan and A. K. Agrawala, "A Decomposition Approach to Nonpreemptive Scheduling in Hard Real-Time Systems", *in Proceedings IEEE Real-Time Systems Symposium*, Dec. 1989.

[11] J. Erschler, G. Fontan, C. Merce, and F. Roubellat, "A New Dominance Concept in Scheduling n Jobs on a Single Machine with Ready Times and Due Dates", *Operations Research*, vol. 31, pp. 114-127, Jan. 1983.

[12] M. R. Garey and D. S. Johnson, *Computers and Intractability, a Guide to the Theory of NP-Completeness*, W. H. Freeman Company, San Francisco, 1979.

[13] M. Dertouzos, "Control Robotics: the Procedural Control of Physical Processes", *Proceedings of the IFIP Congress*, pp. 807-813, 1974.

[14] Al Mok, *Fundamental Design Problems for the Hard Real-Time Environment*, PhD thesis, Massachussets Institute of technology, 1983.

[15] K. Jeffay, D. F. Stanat, and C. U. Martel, "On Non-Preemptive Scheduling of Periodic and Sporadic Tasks", *in Proceedings IEEE Real-Time Systems Symposium*, pp. 129-139, December 1991.

[16] T. Shepard and J. A. M. Gagne, "A Pre-Run-Time Scheduling Algorithm for Hard Real-Time Systems", *IEEE Transactions on Software Engineering*, vol. 17, pp. 669-677, July 1991.

[17] G. McMahon and M. Florian, "On scheduling with ready times and due dates to minimize maximum lateness", *Operations Research*, vol. 23, pp. 475-482, May 1975.

[18] P. Bratley, M. Florian, and P. Robillard, "Scheduling with Earliest Start and Due Date Constraints", *Naval. Res. Log. Quart.*, vol. 18, pp. 511-519, Dec. 1971.

[19] K. R. Baker and Z. Su, "Sequencing with Due-Date and Early Start Times to Minimize Maximum Tardiness", *Naval Res. Log. Quart.*, vol. 21, pp. 171-176, 1974.

[20] J. P. C. Verhoosel, E. J. Luit, D. K. Hammer, and E. Jansen, " A Static Scheduling Algorithm for Distributed Hard Real-Time Systems", *Journal of Real Time Systems*, pp. 227-246, 1991.

[21] G. Fohler and C. Koza, "Heuristic Scheduling for Distributed Real-Time Systems", MARS 6/89, Technische Universitat Wien, Vienna, Austria, April 1989.

[22] W. Zhao and K. Ramamritham, "Simple and Integrated Heuristic Algorithms for Scheduling Tasks with Time and Resource Constraints", *Journal of Systems and Software*, pp. 195-205, 1987.

[23] R. Gerber and W. Pugh and M. Saksena, "Parametric Dispatching of Hard Real-Time Tasks", Technical Report CS-TR-2985, UMIACS-TR-92-118, University of Maryland, Oct. 1992.

[24] C. C. Han and K. J. Lin, "Job scheduling with temporal distance constraints", Technical Report UIUCDCS-R-89-1560, University of Illinois at Urbana-Champaign, Department of Computer Science, 1989.

REPORT DOCUMENTATION PAGE

Form approved
OMB No 074-0188

1. AGENCY USE ONLY (leave blank)	2. REPORT DATE 1/20/1993	3. REPORT TYPE END DATES COVERED Technical Reports

4. TITLE AND SUBTITLE
Temporal Analysis and its Application in Non-Preemptive Scheduling

5. FUNDING NUMNBERS
N00014-91-C-0195

6. AUTHOR(S)
Manas C. Saksena and Ashok K. Agrawala

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Maryland
Department of Computer Science
A.V. Williams Building
College Park, MD 20742

8. PERFORMING ORGANIZATION REPORT NUMBER
CS-TR-2698
UMIACS-TR-91-88

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)
PhillipsLaboratory
Director of Contracting
3651 Lowry Avenue SE
Kirtland AFB, NM 87117-5777

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12.a. DISTRIBUTION/ AVAILABILITY STATEMENT

12.b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
 In the problem of non-preemptive scheduling of a set of tasks on a single processor, each task has a ready time, a deadline and a computation time. A feasible schedule for this set requires that a start time be assigned to each task. The approaches taken for scheduling often use search techniques and may reduce the search by using heuristics. In this paper we present a technique for analyzing the temporal relations among the tasks to establish pairwise relationships among them. These relationships can then be used effectively to reduce the complexity of scheduling these tasks. We present simulation results to confirm the usefulness of temporal analysis as a phase prior to scheduling.

14. SUBJECT TERMS

15. NUMBER OF PAGES
7

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT Unlisted

MSN 7540-01 280-5500

Standard Form 298 (Rev 2-89)

Implementation of the MPL Compiler*†

Jan M. Rizzuto and James da Silva

Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
College Park, MD 20742

February 14, 1995

Abstract

The Maruti Real-Time Operating System was developed for applications that must meet hard real-time constraints. In order to schedule real-time applications, the timing and resource requirements for the application must be determined. The development environment provided for Maruti applications consists of several stages that use various tools to assist the programmer in creating an application. By analyzing the source code provided by the programmer, these tools can extract and analyze the needed timing and resource requirements. The initial stage in development is the compilation of the source code for an application written in the Maruti Programming Language (MPL). MPL is based on the C programming language. The MPL Compiler was developed to provide support for requirement specification. This report introduces MPL and describes the implementation of the MPL Compiler.

1 Introduction

A *real-time* system requires that an application meet the timing constraints specified for it. For *hard real-time*, a failure to meet the specified timing constraints may result in a fatal error [2]. Timing constraints are not as critical for *soft real-time*. The Maruti Operating System was developed to meet the real-time constraints required by many applications. In order to schedule and run an application under Maruti, the timing and resource requirements for that application must be determined. The development environment for Maruti consists of several tools that can be used to extract and analyze these requirements [2].

The *Maruti Programming Language* (MPL) is a language developed to assist users in creating applications that can be run under Maruti. MPL is based on the C programming language, and assumes the programmer is familiar with C. MPL provides some additional constructs that are not part of standard C to allow for resource and timing specification [1]. In addition, when an MPL file is compiled, some of the resource requirements can be recognized and recorded to an output file. This output file is used as input to the integration stage, which is the next stage in the development cycle. During integration, additional timing requirements may be specified.

Previously, an MPL file was compiled by first running the source code through the Maruti pre-compiler, which created a C file that was then compiled using a C compiler [1]. The pre-compiler extracted the necessary information, and converted the MPL constructs that were not valid C statements into C code. This required the additional pass of the pre-compiler over the source code. We have created a compiler for MPL that integrates both the actions of the pre-compiler and the compiler into one stage. In this report, we present MPL, and a description of the compiler we implemented. Section 2 defines the abstractions used in Maruti. In Section 3, the syntax of the constructs unique to MPL is defined. The details of the implementation of the compiler are given in Section 4. Section 5 describes the resource information that is recorded during compilation. Conclusions appear in Section 6, followed by an Appendix containing a sample MPL file, and the resource information recorded for that file.

2 Maruti Abstractions

An MPL application is broken up into units of computation called *elemental units* (EUs). Execution within an EU is sequential, and resource and timing requirements are specified for each EU. A *thread* is a sequential unit of execution that may consist of multiple EUs. MPL allows threads of execution to be specified by the programmer through several of the constructs provided. A *task* consists of a single address space, and threads that execute in that address space. *Modules* contain the source code of the application as defined by the programmer. An application may consist of several modules. During execution, modules are mapped to one or more tasks.

3 MPL Constructs

There are several constructs defined in MPL that are not a part of standard C. These constructs have been implemented in the MPL compiler.

3.1 Module Name Specification

A *module* may consist of one or more source files written in MPL. At the start of each MPL file, the name of the module that the source file corresponds to must be indicated. This is given by the following syntax: .

```
module-name-spec ::= 'module' <module-name> ';'.
```

The `module-name` may be any valid identifier that is accepted by standard C. The module name specification must appear at the beginning of the source file, before any other MPL code. The specification is not compiled into any executable code. It is simply used to indicate the module that the functions within the file belong to.

3.2 Shared Buffers

A *shared buffer* can be used to declare memory that may be shared by several tasks, to permit communication between the tasks. A declaration of a shared buffer requires the type be defined as with a variable declaration. The syntax of a shared declaration is:

```
shared-buffer-decl ::= 'shared' <type-specifier> <shared-buffer-name>.
```

The `shared-buffer-name` can be any valid identifier, and the `type-specifier` can be any valid type for a variable. A shared declaration is compiled as a pointer to the type given in the declaration of the shared buffer, rather than the type given.

3.3 Region Constructs

The are two constructs used to allow for mutual exclusion within an application.

3.3.1 Region Statement

The *region* statement is used to enforce mutual exclusion globally throughout an entire application, and is given by the syntax:

```
region-statement ::= 'region' <region-name>
                      { mpl-statements }.
```

The `mpl-statements` may be any number of valid MPL statements. These statements make up a critical section.

3.3.2 Local Region Statement

The *local_region* statement is used to enforce mutual exclusion within a task, and follows the same syntax of the region statement:

```
local-region-statement ::= 'local_region' <local-region-name>
                            { mpl-statements }.
```

325

3.4 Channel Declarations

Channels are used to allow for message passing within a Maruti application. Each channel declared has a type associated with it given by a valid C type-specifier. This type indicates the type of data that the channel will carry.

Channels may be declared in both *entry* and *service functions*, which will be defined below. The syntax for channel declarations is:

```
channel-declaration-list-opt ::= { channel-declaration-list }.
channel-declaration-list ::= channel-declaration { channel-declaration }.
channel-declaration ::= channel-type channels ';'.
channel-type ::= 'out' | 'in' | 'in-first' | 'in-last'.
channels ::= channel { ',' channel }.
channel ::= <channel-name> ':' type-specifier.
```

3.5 Entry Functions

An *entry function* is a special type of function that may be defined in an MPL source file. Each entry function corresponds to a thread within the application. The syntax for an entry function definition is:

```
entry-function ::= 'entry' <entry-name> '(' ')' entry-function-body.
entry-function-body ::= channel-declaration-list-opt mpl-function-body.
```

3.6 Service Functions

Service functions are another type of special function supported by MPL. A service function is invoked when a message is received from a client. Each service function definition requires an *in* channel and message buffer be included in the definition. The service function will be executed when there is a message on the channel given in the definition. The definition of a service function is similar to that of an entry function:

```
service-function ::= 'service' <service-name>
                '(' <in-channel-name> ':' type_specifier ',' <msg-ptr-name> ')'
                service-function-body.
service-function-body ::= channel-declaration-list-opt mpl-function-body.
```

3.7 Communication Function Calls

There are several library functions used to allow for message passing within a Maruti application.

3.7.1 Send Calls

Each call to the *send* function must specify an outgoing channel for the message:

```
void send ( channel channel_name, void *message_ptr );
```

3.7.2 Receive and Optreceive Calls

Both *receive* calls, and *optreceive* calls must be associated with an incoming channel (*in*, *in_first*, or *in_last*):

```
void receive ( channel channel_name, void *message_ptr );
int optreceive ( channel channel_name, void *message_ptr );
```

A call to receive requires that there be a message on the incoming channel. Optreceive should be used when a message may or may not be on the channel. Optreceive checks for the message, and returns a value indicating if a message was found.

3.8 Initialization Function

Each task has an *initialization routine* that is executed when the application is loaded. This function is specified by the user with the following name and arguments:

```
int maruti_main (int argc, char **argv)
```

4 Implementation

We started with version 2.5.8 of the Gnu C compiler. By modifying the source code for the C compiler, we have created a compiler for applications written in MPL. In addition to what the standard Gnu C compiler does, this modified compiler handles the additional constructs defined in MPL, and records information about the source code that is needed by Maruti. A source code file written in MPL is specified with an *mpl* extension.

4.1 Modifications to GCC File Structure

In the process of modifying the compiler, some existing files were modified. In addition, some new files were also created. The source code for version 2.5.8 of GCC allows compilers to be created for several different languages: C, C++, and Objective C. The GCC compiler uses different executable files for the different languages that it compiles. There are separate files for C, C++, and Objective C *(cc1, cc1plus, cc1obj)*. The GCC driver, *gcc.c*, uses the extension of the source file specified to determine the appropriate executable (and therefore language) to compile the source file. The driver then executes the compiler, passing on the appropriate switches. The driver was modified to accept input files with an *mpl* extension. *Cc1mpl* is the new executable that was created to compile MPL source files. When a file with an *mpl* extension is specified as a source file to be compiled, this new executable file is used. When an MPL file is compiled, it automatically passes on the switch *-Maruti_output*, which indicates that the needed output should be recorded to a file with an *eu* extension.

The executable files for each language are composed of many object files. Some of these files are common to all the languages, and some of the files are language-specific. The language-specific files added for compiling MPL files are those files with an *mpl-* prefix.

Gperf is a tool used to generate a perfect hash function for a set of words. Gperf is used to create a hash function for the reserved words for each language. The files containing the input to gperf are indicated by a file name with a *gperf* extension. There are several different **.gperf* files containing the reserved words for the different languages recognized by

327

the compiler. The *mpl-parse.gperf* file contains all the reserved words for C, in addition to those added for MPL. For each language, the output from running gperf is then incorporated into the *-lex.c* file. This output includes a function *is_reserved_word()* that is used to check if a token is a reserved word. The file *mpl-lex.c* is basically the *c-lex.c* file, with the output of running gperf on *mpl-parse.gperf* instead of *c-parse.gperf*.

The file *maruti.c* contains the routines that have been written to implement MPL. This file is linked in with the executable for all of the languages, to prevent undefined symbol errors from occuring. Calls to the routines contained in this file occur in both the language-specific, and the common files. The flag *maruti_dump* is set in *main()* to indicate whether information about the source code should be recorded to the appropriate output file. This flag prevents calls to the routines in *maruti.c* which are made in the common files from occuring for the languages other than MPL. The files containing these calls are:

- *calls.c*
- *explow.c*
- *expr.c*
- *function.c*
- *toplev.c*

There are several reasons why the new language-specific files have to be created for MPL. The files *mpl-lex.h* and *mpl-lex.c* needed to be created for MPL because MPL contains several additional reserved words not present in C, as mentioned earlier. The file *c-common.c* relies on information in the header file *c-lex.h*. Since MPL uses *mpl-lex.h*, *mpl-common.c* includes *mpl-lex.h*, instead of *c-lex.h*. *Bison* is a tool that allows a programmer to define a grammar through rules, and converts them into a C program that will parse an input file. The *-parse.y* files are the bison files used to create the grammar to parse a source file. Since the grammar for MPL needed to be modified to accept the additional constructs, the *mpl-parse.y* file was created. There is one function used in compiling MPL source files that is defined in *mpl-parse.y*, instead of *maruti.c*. This function needed to access the static variables declared in *mpl-parse.y*, and in order to do so, the function definition was placed in that file. Finally, the file *mpl-decl.c* was created, because of its dependence on *mpl-lex.h*, and also to allow for an additional type specification used in MPL.

4.2 Compiling MPL Constructs

MPL extends the C language to allow for various constructs. In order to implement these extensions, the grammar used to recognize C in GCC had to be extended. The following are recognized as reserved words for MPL, in addition to the standard reserved words for C: *shared, region, local_region, module, in, out, in_first, in_last, entry, service, send, receive,* and *optreceive*. The keywords *in* and *out* were reserved words in the *c-** files, because they are used by Objective C, but in MPL they are used as channel types. In addition to the new reserved words, rules were added and modified resulting in the rules in *mpl-parse.y*.

4.2.1 Module Name Specification

A rule was added to the grammar to parse the module name specification in an MPL file. The rule for a whole program was also modified to include this module statement. This rule expects the module statement to appear before any other definitions. Since the module

name specification does not result in any executable code, the only action taken is to record the module name given by the programmer.

4.2.2 Shared Buffers

There are no rules added to the grammar for a shared buffer declaration. When a variable declaration is parsed, a tree is created that keeps track of all the specification information given for that declaration. For example, *typedef* and *extern* are two of the possible type specifications. The token *shared* is recognized as a type specification, just as *typedef* and *extern* are recognized. When a declaration is made, these specifications are processed in the function *grokdeclarator()* in *mpl-decl.c*. When a shared specification is encountered, the declaration is converted to a pointer to the type specified, instead of just the type specified. Other than this conversion to a pointer, the declaration is compiled just as any other declaration would be compiled in C.

4.2.3 Region Constructs

The region constructs are considered statements in MPL. Several rules were added to parse these constructs, and the *region* and *local_region* statements were added as options for a valid statement in the grammar for MPL.

Both region and local_region statements are compiled in the same manner. Each region has a name, and a body which is the code within the critical section. In order to protect these critical sections, calls are made to the Maruti library function *maruti_eu()*. When a region is parsed, the compiler generates two calls to *maruti_eu()*, in addition to the code in the body of the region. The first call is generated just before the body, and the second call just after. These calls are generated through functions in *maruti.c*. The functions are based on the actions that would have been taken, had the parser actually parsed the calls to *maruti_eu()* in the source file.

4.2.4 Channel Declarations

The rules added for a channel declaration allow any number of channels to be declared in either an entry or a service function. Each channel declaration requires several pieces of information:

- *Channel-type*
- *Channel-name*
- *Type specifier indicating the type of data that channel carries*

A linked list of declared channels is maintained. For each declared channel the following information is saved:

- *Channel-name*
- *Type information*
    1. *Size in bytes*
    2. *String encoding the type of the data*
- *Channel-id*

The *channel-id* is a unique identification number assigned to each declared channel. Channel declarations do not add to the compiled code. The channels are not allocated memory. The information describing each channel is simple stored in the linked list. During compilation, whenever a channel is referenced, the appropriate information is obtained from this list.

4.2.5 Entry Functions

Entry function definitions are compiled differently than other function definitions. An entry function would appear in an MPL file in the following form:

```
entry <entry_name> ()
<channel_declaration_list_opt>
{
  <mpl_function_body>
}
```

Where `entry_name` is an identifier that is the name of the entry function, the `channel_declaration_list_opt` contains any channels the user wants to define for that function, and `mpl_function_body` is any function body that would be accepted as a definition in a standard MPL function. Semantically the entry function is equivalent to the following MPL code:

```
_maruti_entry_name ()
{
  while(1)
  {
    maruti_eu();
    entry_name () ;
  }
}

entry_name ()
{
  mpl_function_body
}
```

An entry function is compiled into two functions, as if the two functions given above had been part of the source file. Essentially, the first function is just a stub function that calls *maruti_eu()*, then calls the second function compiled. As with generating function calls, the routines to generate the code for entry function definitions are based on the actions that would have been taken had the parser actually parsed the code for the two separate functions.

4.2.6 Service Functions

Service functions definitions are handled very much like entry function definitions. The syntax of a service function differs slightly from that of an entry function, since it requires that an incoming channel and a message buffer be defined:

330

```
service <service_name> (<in_channel_name> : <type_specifier>, <msg_ptr_name>)
<channel_declaration_list_opt>
{
    <mpl_function_body>
}
```

Like the entry functions, service functions are semantically equivalent to two functions,
where one is simply a stub function calling the second function that is generated:

```
_maruti_service_name ()
{
type_specifier _maruti_msg_ptr_name ;

  while(1)
  {
    if ( optreceive ( _maruti_in , id  , & _maruti_msg_ptr_name, size ) )
    {
        service_name (& _maruti_msg_ptr_name );
    }
  }
}


service_name (msg_ptr_name)
type_specifier  *msg_ptr_name;
{
  mpl_function_body
}
```

The service_name, channel_declaration_list, and mpl_function_body are all the same
as described previously for entry functions. In addition, service functions have two other
items specified in their definitions. The first is a channel. Every service function requires
a channel be specified. This channel is always declared as an *in* channel with the name
in_channel_name. The type is given by type_specifier as if it had been declared in the
channel_declaration_list. The channel is used to invoke the service function. This in
channel is used by the optreceive in the stub function that calls the function containing the
service function body. When a message is received on this channel, the service function is
executed. The second additional item is a message buffer used by the service function. The
name of this message buffer is given by msg_ptr_name, the type is given by type_specifier.
This buffer is used to hold the message received from the client that invoked the service
function, and is passed to the second function containing the body of the service function.

4.2.7 Communication Function Calls

There were three library functions provided for message passing mentioned previously: send,
receive, and optreceive. Function calls to any of these three library functions are handled
differently than other function calls. In the MPL grammar, *send, receive,* and *optreceive*
are all reserved words. The MPL syntax for all of these calls is the following:

```
<function-name> (<channel-name>, <parameter-2>);
```

331

`Channel-name` should be a previously declared channel, and `parameter-2` should be a pointer. These function calls must be compiled differently, since these are not the actual parameters used when the call is generated. In the case of a call to send, the actual parameters must be as follows:

```
send (<channel-id>, <parameter-2>, <channel-size>);
```

In the case of a call to either receive, or optreceive, the parameters required are:

```
receive | optreceive (<channel-type>, <channel-id>, <parameter-2>, <channel-size>);
```

The `channel-type` for a receive or optreceive call is an integer generated by the compiler that will indicate an *in, in_first*, or *in_last* channel.

When one of these three function calls are encountered, there are special rules in the grammar to handle it. A function in *maruti.c* is called which generates the appropriate parameters, and then the function call itself. These function calls are generated as mentioned above for the calls to *maruti_eu()*. The `channel-name` specified by the user is used to obtain the necessary parameters. Given the channel name, the linked list of channels is searched to find the corresponding channel, then the `channel-id` and the `channel-size` are obtained from that node in the linked list. There is also some type checking done at this stage. The compiler verifies that only an outgoing channel is specified for a send call, or an incoming channel for the receive and optreceive calls. The compiler also checks that any channel referenced has been previously defined.

The grammar for MPL was modified so that a call to any of the communication functions may occur anywhere that a primary expression occurs, since that is where other function calls are permitted to occur.

4.2.8 Initialization Function

The user-defined function *maruti_main()* is compiled as an ordinary C function.

5 PEUG File

The source code of an MPL file is broken up into *elemental units*. Each elemental unit identifies the resources that it requires. These elemental units are used later in the development process for scheduling the application. The output file created by the MPL compiler creates a *Partial Elemental Unit Graph* (PEUG) for the given source file. The name of this file is the name of the source file, with the *mpl* extension replaced by an *eu* extension.

There are several different types of information recorded in this PEUG file.

5.1 Module Name

The first line in the output file indicates the name of the module, and will appear as:

```
peug <module-name>
```

The `module-name` is taken directly from the module name specification given in the MPL source file.

5.2 File Name

The second line in the source file indicates the name of the target file that is created by the compiler, where file-name is the target:

```
file <file-name>
```

5.3 Shared Buffers

Each time a shared buffer is declared its name and type information is recorded to the output file:

```
shared <shared-buffer-name> : (type-description-string>, <type-size>)
```

The type-description-string and type-size of a shared buffer is obtained from the type specification, and is represented in the same manner as the type and size for a channel. Although the shared buffer is actually a pointer to the type it is declared as, the type-description-string represents the object being pointed to, and not the pointer itself.

5.4 Entry, Service, and User Function Definitions

In MPL, a user may define ordinary functions in addition to the entry and service functions that are permitted in MPL. For each entry, service or ordinary user-defined function, there is an entry in the output file. This entry has the following format:

```
<function-type> <function-name>
      .
      .
      .
      size <stack-size>
```

Function-type can be either function, entry, or service, indicating which type of function is being defined. Function-name is the declared name of the function in the source file. Stack-size is the maximum stack size needed by this function. This stack-size includes the arguments pushed onto the stack preceding any function calls occuring within the function body. There will also be other information concerning the body of the function that will appear between the function-name, and the stack-size. The entry for the *maruti_main()* function will be the same as those for other user defined functions. Entry and service functions will contain some additional information not applicable to ordinary functions that will be described below.

5.4.1 Channels

For each channel that is declared, a description of the channel is written to the output file. These descriptions will occur right after the statement indicating the name of the current function:

```
<channel-type> <name> : (<description-string>, <size>)
```

333

The channel-type and channel-name will be the type and name specified in the source file. The description-string and size are based on the type specification in the channel declaration. Channel descriptions will occur only in entry and service functions. A service function will always contain at least one channel description, since the syntax of a service function requires a channel be named in the definition. A channel description will also be output for every send, receive, and optreceive call, since these calls require a channel as one of their parameters.

5.4.2 Function Calls

Each time a function call is parsed, there will be a line in the output file:

```
calls <function-name> {in_cond} {in_loop}
```

This line indicates where a function call occurs, and which function is being called. The in_cond and in_loop indicate if this function call appears within a conditional statement or within a loop. These labels will be seen only if their respective conditions are true.

5.4.3 Communication Function Calls

Any call to a communication function is recorded similarly to other function calls. There is a line indicating the name of the function, as shown above for a function call. In addition, there will be a line describing the channel associated with that communication function call. This line will appear just as the line for the channel definition described above appears.

5.4.4 EU Boundaries

The output file for an MPL source file indicates where each elemental unit (EU) begins by the following:

```
eu <N> {region_list}
```

The N indicates an EU number. Each EU within a source file has a unique number. There are several places where EU boundaries are created:

- *Start of a function*
- *Start of a region*
- *End of a region*
- *Explicit calls to maruti_eu()*

The initial EU occuring at the beginning of a function that is not a service or entry function is a special case. This is always labeled as "eu 0" in the output file, and does not represent an actual EU.

Each EU may also be followed by a list describing one or more regions. This list represents the regions that this EU occurs within. The description of a region appears as:

```
(region-name instance access type)
```

The region-name is just that given by the user, and the type indicates if a region is *local* (local_region construct) or *global* (region construct). The access indicates if the access is read or write. The instance indicates the instance of this region within the source file. Each instance for a region within a source file is unique.

334

6 Conclusions

Basing MPL on C has simplified the development of both the language and its compiler. The language is easy to learn for any programmer that has used C before, since there are a limited number of additional constructs unique to MPL. Using the GCC C source code provided an existing compiler, rather than implementing a new one. The source code for GCC only needed to be modified to handle some additional constructs, and produce some additional output. This made the implementation fairly simple. However, the GCC C compiler also provides some functionality that is not needed by MPL. Much of this functionality provided is not even permitted. These restrictions are not enforced by the compiler, but should be detected within the development cycle.

Prior to the development of the MPL compiler using GCC, compiling an MPL source file required two steps. The source files were initially passed through a pre-compiler to extract the available resource information and parse the MPL constructs. The pre-compiler was responsible for converting the MPL code into valid C code, which was then compiled using a standard C compiler. The new implementation of the compiler eliminates some of the redundant processing that is done when the pre-compiler is used. The information obtained through the pre-compiler already existed in the internal structure used by the GCC compiler. This information just needed to be recorded. Instead of parsing source code files in the two steps independently, the functionality of the pre-compiler has been incorporated into the compiler itself. The MPL compiler provides a single tool that extracts all the available information at the initial stage of develpment.

In the future, a version of MPL may be implemented that is based on the Ada programming language. GNAT is a compiler for Ada 9X that is being developed at NYU. GNAT depends on the backend of the GCC compiler. Using the source code for GNAT, an implementation of MPL based on Ada would be similar to the current implementation based on C.

Appendix

A MPL File

The following is a sample of MPL source code:

```
module timer;

typedef struct {
    int seconds;
    int minutes;
    int hours;
    } time_type;

shared time_type global_time;

maruti_main(argc, argv)
int argc;
char **argv;
{
global_time->seconds = 0;
global_time->minutes = 0;
global_time->hours = 0;

return 0;
}


entry update_second()
out disp : time_type;
{
time_type msg;

region time_region {
    global_time->seconds++;
    if (global_time->seconds == 60)
        global_time->seconds = 0;
    msg = *global_time;
    }

send (disp, &msg);
}

entry update_minute()
out display : time_type;
{
time_type msg;

region time_region {
    global_time->minutes++;
```

```
    if (global_time->minutes == 60)
        global_time->minutes = 0;
    msg = *global_time;
    }

send (display, &msg);
}

entry update_hour()
out display : time_type;
{
time_type msg;

region time_region {
    global_time->hours++;
    if (global_time->hours == 24)
        global_time->hours = 0;
    msg = *global_time;
    }

send (display, &msg);
}

service display_time(inchan : time_type, time)
{
printf("Current Time: %d : %d : %d", time->hours, time->minutes, time->seconds);
}
```

B PEUG File

The corresponding PEUG file for the source code above is:

```
peug timer
file timer.o
shared global_time : ($(iii), 12)
function maruti_main
        eu 0
        size 4
entry update_second
        out disp : ($(iii), 12)
        eu 2
        eu 3 (time_region 1 W global)
        calls maruti_eu
        eu 4
                calls maruti_eu
                calls send
                out disp : ($(iii), 12)
        size 32
entry update_minute
        out display : ($(iii), 12)
        eu 5
        eu 6 (time_region 2 W global)
                calls maruti_eu
        eu 7
                calls maruti_eu
                calls send
                out display : ($(iii), 12)
        size 32
entry update_hour
        out display : ($(iii), 12)
        eu 8
        eu 9 (time_region 3 W global)
                calls maruti_eu
        eu 10
                calls maruti_eu
                calls send
                out display : ($(iii), 12)
        size 32
service display_time
        in inchan : ($(iii), 12)
        eu 11
                calls optreceive
                in inchan : ($(iii), 12)
                calls printf
        size 52
```

References

[1] James da Silva, Eric Nassor, Seongsoo Hong, Bao Trinh, and Olafur Gudmundsson. Maruti 2.0 Programmer's Manual. Unpublished.

[2] Manas Saksena, James da Silva, and Ashok Agrawala. Design and Implementation of Maruti-II. In Sang H. Son, editor, *Advances in Real-Time Systems*, chapter 4. Prentice Hall, 1995.

[3] Richard Stallman. The GNU C compiler, version 2.5.8., Manual. Info file obtained from gcc.texi in source code distribution.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	February 14, 1995	Technical Report

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Implementation of the MPL Compiler	N00014-91-C-0195 and DSAG-60-92-C-0055

6. AUTHOR(S)

Jan M. Rizzuto and James da Silva

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
University of Maryland A.V. Williams Building College Park, Maryland 20742	CS-TR-3413 UMIACS-TR- 95-17

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSORING/MONITORING AGENCY REPORT NUMBER
Honeywell 3660 Technology Drive Minneapolis, MN 55418	Phillips Labs 3550 Aberdeen Ave. SE Kirtland AFB, NM 87117-5776	

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The Maruti Real-Time Operating System was developed for applications that must meet hard real-time constraints. In order to schedule real-time applications, the timing and resource requirements for the application must be determined. The development environment provided for Maruti applications consists of several stages that use various tools to assist the programmer in creating an application. By analyzing the source code provided by the programmer, these tools can extract and analyze the needed timing and resource requirements. The initial stage in development is the compilation of the source code for an application written in the Maruti Programming Language (MPL). MPL is based on the C programming language. The MPL compiler was developed to provide support for requirement specification. This report introduces MPL and describes the implementation of the MPL Compiler.

14. SUBJECT TERMS			15. NUMBER OF PAGES
D.3.2, Language Classifications D.4.7, Organization and Design			17 pages
			16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited

Maruti 3.1
Programmer's Manual
First Edition

Systems Design and Analysis Group
Department of Compter Science
University of Maryland at College Park

December 1996

Contents

Chapter 1

Introduction

The *Maruti Programming Language* (MPL) is used to write Maruti application code. Currently, MPL is based on the ANSI C programming language, with extensions to support modules, real-time constructs, communications primitives, and shared memory.

The *Maruti Configuration Language* (MCL) is used to specify how individual program modules are to be connected together to form an application and the details of the hardware platform on which the application is to be executed.

1.1. General Program Organization

A complete Maruti system is called an *application*. Applications can be large, distributed systems made up of many subsystems. Each application is defined by a *configuration file*, which defines all the subsystems and their interactions. The following entities make up an application:

Jobs Jobs are the active entities in a Maruti application. Jobs are specified in the configuration file with timing constraints, including the job period. A job is made up of multiple *entry points*, which are the threads of execution that will be run for the job.

Modules The code of an application is divided into modules. Each module consists of *entry points*, which define the code which will be executed as part of a job, *services*, which define code to be invoked on behalf of a client module, and *functions*, which are called from entries and services.

Tasks At run-time, modules map to tasks (a module may be mapped to more than one task). Each task consists of an *address spac* and *threads* of execution for the entry points and services of the module.

Channels Channels are the communication paths for Maruti applications. Each channel is a one-way connection through which typed messages are passed. The end points are defined by *out* and *in* channel specifiers, and are connected as specified in the application configuration file. Each end point is associated with one entry or service, and its message type and channel type are declared within the entry or service header. The types of the in and out channel specifiers must match.

Regions Regions are the mechanism for mutual exclusion between Maruti threads: only one thread can enter a particular region at a time. Two types of regions may be specified: *global* regions enforce exclusion for the entire Maruti application, while *local* regions enforce exclusion only within a single task.

Shared buffers Named memory buffers can be shared between tasks. The buffer is mapped into the address space of each task that uses that buffer.

1.1.1. Maruti Programming Language

Rather than develop completely new programming languages, we have taken the approach of using existing languages as base programming languages and augmenting them with Maruti primitives needed to provide real-time support.

In the current version, the base programming language used is ANSI C. MPL adds *modules, shared memory blocks, critical regions, typed message passing, periodic functions*, and *message-invoked functions* to the C language. To make analyzing the resource usage of programs feasible, certain C idioms are not allowed in MPL; in particular, recursive function calls are not allowed nor are unbounded loops containing externally visible events, such as message passing and critical region transitions.

- The code of an application is divided into *modules*. A module is a collection of procedures, functions, and local data structures. A module forms an independently compiled unit and may be connected with other modules to form a complete application. Each module may have an *initialization function* which is invoked to initialize the module when it is loaded into memory. The initialization function may be called with arguments.

- Communication primitives send and receive messages on one-way, typed *channels*. There are several options for defining channel endpoints that specify what to do on buffer overflow or when no message is in the channel. The connection of two endpoints is done in the MCL specification for the application - Maruti insures that endpoints are of the same type and are connected properly at runtime.

- Periodic functions define *entry points* for execution in the application. The MCL specification for the application will determine when these functions execute.

- Message-invoked functions, called *services*, are executed whenever messages are received on a channel.

- *Shared memory blocks* can be declared inside modules and are connected together as specified in the MCL specifications for the application.

- *Critical Regions* are used to safely maintain data consistency between executing entities. Maruti ensures that no two entities are scheduled to execute inside their critical regions at the same time.

1.1.2. Maruti Configuration Language

MPL Modules are brought together into as an executable application by a specification file written in the Maruti Configuration Language (MCL). The MCL specification determines the application's hard real-time constraints, the allocation of tasks, threads, and shared memory blocks, and all message-passing connections. MCL is an interpreted C-like language rather than a declarative language, allowing the instantiation of complicated subsystems using loops and subroutines in the specification. The key features of MCL include:

- **Tasks, Threads, and Channel Binding.** Each module may be instantiated any number of times to generate tasks. The threads of a task are created by instantiating the entries and services of the corresponding module. An entry instantiation also indicates the job to which the entry belongs. A service instantiation belongs to the job of its client. The instantiation of a service or entry requires binding the input and output ports to a channel. A channel has a single input port indicating the sender and one or more output ports indicating the receivers. The configuration language uses channel variables for defining the channels. The definition of a channel also includes the type of communication it supports, i.e., synchronous or asynchronous.

- **Resources.** All global resources (i.e., resources which are visible outside a module) are specified in the configuration file, along with the access restrictions on the resource. The configuration language allows for binding of resources in a module to the global resources. Any resources used by a module which are not mapped to a global resource are considered local to the module.

- **Timing Requirements and Constraints.** These are used to specify the temporal requirements and constraints of the program. An application consists of a set of cooperating jobs. A job is a set of entries (and the services called by the entries) which closely cooperate. Associated with each job are its invocation characteristics, i.e., whether it is periodic or aperiodic. For a periodic job, its period and, optionally, the ready time and deadline within the period are specified. The constraints of a job apply to all component threads. In addition to constraints on jobs and threads, finer level timing constraints may be specified on the observable actions. An observable action may be specified in the code of the program. For any observable action, a ready time and a deadline may be specified. These are relative to the job arrival. An action may not start executing before the ready time and must finish before the deadline. Each thread is an implicitly observable action, and hence may have a ready time and a deadline.

Apart from the ready time and deadline constraints, programs in Maruti can also specify *relative* timing constraints, those which constrain the interval between two events. For each action, the start and end of the action mark the observable events. A relative constraint is used to constrain the temporal separation between two such events. It may be a relative deadline constraint which specifies the upper bound on time between two events, or a delay constraint which specifies the lower bound on time between the occurrence of the two events. The interval constraints are closer to the event-based real-time specifications, which constrain the minimum and/or maximum distance between two events and allow for a rich expression of timing constraints for real-time programs.

- **Replication and Fault Tolerance.** At the application level, fault tolerance is achieved by creating resilient applications by replicating part, or all, of the application. The configuration language eases the task of achieving fault tolerance, by allowing mechanisms to replicate the modules, and services, thus achieving the desired amount of resiliency. By specifying allocation constraints, a programmer can ensure that the replicated modules are executed on different partitions.

Chapter 2

Tutorial

2.1. Basic Maruti Program Structure

Maruti applications are built up out of one or more MPL modules, and tied together with a *configuration file* written in MCL. We'll start our tutorial with an explanation of a very simple application consisting of one module, called simple.mp. Our simple application will contain a producer thread that sends out integer data, and a consumer thread, which receives integer values and prints them out.

The Module

```
module simple;

int data;

maruti_main(int argc, char **argv)
{
    if(argc < 1) {
        printf("simple: requires an integer argument\n");
        return 1;
    }

    data = atoi(argv[0]);
    return 0;
}
```

This first part of the module will be similar in all Maruti modules. The module always starts with the module name declaration. After the module declaration, the MPL module is much like any ANSI C program, but with some special Maruti definitions.

Every module must contain a function named maruti_main, which initializes the module at load time. This initialization would normally include things like device probing or painting the screen. The maruti_main function, exactly like the main function of a C program, takes an argument count and list as its parameters, and returns an error code to its environment. In Maruti, the environment is the system loader, and any non-zero return results in a load failure, in which case the application will not run. In our example, maruti_main is responsible for setting the initial value of our datum from the environment, and returning a failure code if there is no argument .

347

Periodic Functions

```
entry producer()
      out och: int;
      {
            data++;                    /* produce data */
            send(och, &data);
      }
```

The producer is a periodic function, or Maruti *entry point*. It serves as the top-level function for a Maruti thread that will be invoked repeatedly, with a period specified in the MCL config file (which we will see below).

The producer outputs its data on a Maruti *channel*, using the built in MPL send function. The channel och is declared as part of the function header of producer. Maruti channels are declared to have a type, usually a structure but in this case a simple integer. All messages sent on the channel will be of the same type.

Note that there is no open, bind, or connect statement needed to initiate communication on the channel. The connection of the channel will be specified in the config file, and initiated automatically by the runtime system.

Message-Invoked Functions

```
service consumer(ich: int, msg)
{
      printf("consumer got %d\n", *msg);    /* consume data */
}
```

The consumer is a message-invoked function, or Maruti *service*. It serves as the top-level function for a Maruti thread that is invoked whenever there is a message delivered on the channel declared in the function header. The msg parameter is the name of the pointer to the message buffer that will contain the delivered message.

Since the receipt of the invoking message is automatic for a Maruti service, the only thing our consumer has to do is print out the data value contained in the message.

This completes our simple module, but in order to have a Maruti application, we must have a config file that tells the system how to run our program.

The Config File

The config file is written in the Maruti Configuration Language (MCL), an interpreted C-like language with constructs that allow an application to be built up from pieces and interconnected. The MCL processor, called the *integrator*, builds a program graph from the specifications, analyses it for type correctness and completeness, and checks for dependency cycles. Here is the config file, *simple.cfg*, that goes with our application :

```
application simple {

        job j;                 /* declare variables */
        task si;
        channel c;

        init j: period 1 s;    /* specify job parameters */
        start si: simple(27);    /* specify task parameters */

        <> si.producer <c> in j;    /* producer thread */
        <c> si.consumer <>;        /* consumer thread */
}
```

The variables in MCL correspond to the objects that make up an application, such as channels, tasks, and jobs. As in C, these variables must be declared before they are used.

In Maruti, a *job* is a logical collection of threads that run with the same period. All *entry* functions in the application must be put *in* some job. The *init* statement sets the period for a particular job. In our case, the job *j* will run once every second.

A *task* is the runtime instantiation of an MPL module, just as in Unix a process is the runtime image of a program. Many tasks may be executed from the same module, each will run independently in the Maruti application. The MCL start command instantiates a task from a module. In our example, we instantiate one task from the module *simple* and pass it the initial data value of *27*.

We instantiate the threads for the *entry* and *service* functions inside a particular task, with particular input and output channels. In our example, the statement

        <> si.producer <c> in j;   /* producer thread */

instantiates the *si.producer* thread in job j with no input channels and one output channel, *c*. Likewise, the statement

        <c> si.consumer <>;   /* consumer thread */

instantiates the *si.consumer* thread with one input channel *c*, and no output channels. Service functions are not put in a job, but rather inherit the scheduling characteristics of the thread that is sending to their invoking channel.

The *integrator* checks to insure that the use of *producer* and *consumer* in the config file match the declarations in the program module.

Building and Running the Application

We can build the *simple* application by putting *simple.mpl* and *simple.cfg* in a directory, and running the *mbuild* command there:

```
% ls
simple.cfg    simple.mpl
% mbuild
mbuild: extracting module info from MCL file 'simple.cfg'
mbuild: creating obj subdirectory for output files.
mbuild: generating obj/simple-build.mk
mbuild: running make -f obj/simple-build.mk

...
```

Mbuild takes care of running the MPL compiler, the MCL integrator, as well as the analysis and binding programs needed to build the runnable Maruti application. By default, mbuild creates both a stand-alone binary that can be booted on the bare machine, and a Unix binary that runs in virtual real time from within the Unix development environment. These different versions of the runtime system are called *flavors*.

We can try out the simple application by running the *ux+x11* flavor from the command line:

```
% obj/simple.ux+x11
<... startup messages ...>
consumer got 28
consumer got 29
consumer got 30
consumer got 31
consumer got 32
consumer got 33
consumer got 34
consumer got 35
consumer got 36
consumer got 37

...
application quit
```

The application boots up and outputs the consumer message once every second. We can exit the application by typing '*q*'.

2.2. Using the Graphics Library

Many Maruti programs will want to use the graphical screen as a monitor for an embedded system, producing oscilloscope or bar-graph style displays, or for animating a simulation or demonstration. Maruti provides a console graphics library as an integral part of the system to make the development of visually oriented applications simpler. Our next example application, *clock*, demonstrates the use of the graphics library as well as the use of multiple jobs to take advantage of Maruti's scheduling abilities.
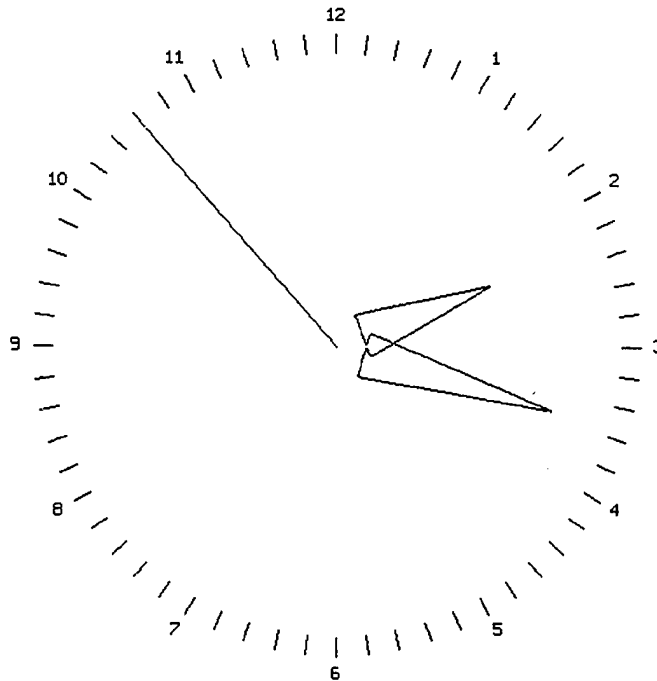
Figure 2.1: Dsiplay of clock example application

The *clock* application will display a circular clock face on the screen, with the hour, minute, and second hands moving as independent Maruti threads in different jobs. The clock screen is shown in Figure 2.1.

We will now go through the clock.mpl module and see how it works.

```
module clock;

#include <maruti/mtime.h>
#include <maruti/console.h>
#include <math.h>

#include "clock.h"

#define CENTER_X    (CONSOLE_WIDTH/2)    /* useful constants */
#define CENTER_Y    (CONSOLE_HEIGHT/2)

void check_for_quitkey(void);            /* subroutines */
void polar_point(int pos, int radius, int *x, int *y);
void xor_triangle(int pos, int apex_radius,int color);
void xor_ray(int pos, int color);

int sec_pos, min_pos, hour_pos;          /* system state */
```

The first part of the module is much like any other ANSI-C program, with *#* *includes, # defines,* and function prototype declarations for subroutines to be used later in the program. Notice the two Maruti header files included: <maruti/mtime.h> contains declarations related to Maruti time management, and <maruti/console.h> contains declarations that define the graphics library interface. The *"clock.h"* header, which we'll see below, will contain definitions that customize the look of the clock face.

```
maruti_main()
{
    int i, x1, y1, x2, y2, color;
    char num_str[4];
    mtime curtime;
    mdate curdate;

    /* initialize screen library, paint screen black */

    cons_graphics_init();
    cons_fill_area(0, 0, CONSOLE_WIDTH, CONSOLE_HEIGHT, BLACK);
```

The maruti_main function in the clock application draws the clock face display and initializes the system state - in our case, the positions of the three clock hands. Before drawing on the screen, the application must call cons_graphics_init, and initialize the contents of the screen. The call to cons_fill_area does this by filling the entire screen with the color BLACK.

```
/* draw tick marks for clock face */

for(i = 0; i < 60; i++) {
    polar_point(i*POS_PER_TICMARK, OUTER_RADIUS, &x1, &y1);
    polar_point(i*POS_PER_TICMARK, INNER_RADIUS, &x2, &y2);

    if(i % 5) color = GRAY;
    else color = WHITE;

    cons_draw_line(x1, y1, x2, y2, color);
}
```

The step in initialization is to draw the tick marks for the clock face. There will be sixty tick lines drawn around the circle, one for each second. Every fifth tick mark will be WHITE to mark the hour positions, and the rest will be GRAY. The lines are drawn using the cons_draw_line library routine, which draws a one-pixel-wide line between two points in the desired color.

The location of the endpoints of our tick marks are calculated using a helper routine, polar_point (shown below), which calculates the cartesian coordinates for a given angle

and radius. We conveniently adopt integer angle positions starting from 0 at the top, clockwise around up to 60*POS_PER_TICMARK back at the top again.

```
/* draw numerals for clock face */

for(i = 1; i <= 12; i++) {
    sprintf(num_str, "%d", i);
    polar_point(i*5*POS_PER_TICMARK, NUMBER_RADIUS, &x1, &y1);
    y1 -= 8; x1 -= strlen(num_str)*8 / 2;   /* center the string */
    cons_print(x1, y1, num_str, strlen(num_str), YELLOW);
}
```

The numerals are placed on the clock face similarly to the tick marks. The cons_print graphics library function places text on the screen at a given position and color.

```
/* initialize the hand positions to current time */

maruti_get_current_time(&curtime);
curdate = maruti_time_to_date(curtime);


sec_pos = curdate.second * POS_PER_TICMARK;
min_pos = curdate.minute * POS_PER_TICMARK +
        curdate.second * POS_PER_TICMARK / 60 ;
hour_pos = (curdate.hour % 12) * 5 * POS_PER_TICMARK +
        curdate.minute * 5 * POS_PER_TICMARK / 60;


return 0;
}
```

The final part of the initialization is the calculation of the initial placement of the clock hands. The maruti_get_current_time|system call returns the current system time, given as a mtime structure. The system time is kept just as in Unix—as the number of seconds and microseconds since the *Epoch* time, defined as 00:00 GMT on January 1, 1970. The maruti_time_to_date library routine does the job of calculating the date and time-of-day from an mtime value.

```
entry sec_hand()
{
    static int erase = 0;


    if(erase) xor_ray(sec_pos, WHITE);
    else erase = 1;
```

```
    sec_pos = (sec_pos + POS_PER_TICMARK) % NUM_POSITIONS;
    xor_ray(sec_pos, WHITE);


    check_for_quitkey();
}
```

The periodic function sec_hand will be run once per second. It erases the previously placed second-hand ray, calculates the new position and draws again there. The check_for_quitkey subroutine (shown below) will poll the keyboard and exit the application if a key is pressed.

```
entry min_hand()
{
    static int erase = 0;


    if(erase) xor_triangle(min_pos, MIN_RADIUS, MIN_COLOR);
    else erase = 1;


    min_pos = (min_pos + 1) % NUM_POSITIONS;
    xor_triangle(min_pos, MIN_RADIUS, MIN_COLOR);
}

entry hour_hand()
{
    static int erase = 0;


    if(erase) xor_triangle(hour_pos, HOUR_RADIUS,HOUR_COLOR);
    else erase = 1;


    hour_pos = (hour_pos + 1) % NUM_POSITIONS;
    xor_triangle(hour_pos, HOUR_RADIUS, HOUR_COLOR);
}
```

The min_hand and hour_hand periodic functions update their respective hand positions by one each time they are called. The second hand jumps forward one second each time it is called, but the minute and hour hands creep forward in smaller relative increments (rather than jumping forward once per minute or hour, which would not look right).

```
    void polar_point(int pos, int radius, int *x, int *y)
    {
        double angle = (2.0*M_PI/NUM_POSITIONS) * (NUM_POSITIONS-pos) +
M_PI/2;
```

```
    *x = CENTER_X + cos(angle) * radius;
    *y = CENTER_Y - sin(angle) * radius;
}
```

Finally we come to the helper functions. The polar_point function converts from our convenient "positions" to real angles in radians, taking into account that radians start at the right and run counter-clockwise, whereas our positions start at the top and run clockwise. Given an angle in radians and a radius from the center, the x and y coordinates of the point are found by taking the cosine and sine of the angle. The final twist is that in cartesian coordinates, the y axis points up, whereas in screen coordinates it traditionally points down, so the y coordinate must be flipped around.

```
void xor_ray(int pos, int color)
{
    int x, y;
    polar_point(pos, SEC_RADIUS, &x, &y);
    cons_xor_line(CENTER_X, CENTER_Y, x, y, color);
}
void xor_triangle(int pos, int apex_radius, int color)
{
    int xb1, yb1, xb2, yb2, xp, yp;
    int bp1, bp2;
    bp1 = (pos + TRIANGLE_BASEL/2) % NUM_POSITIONS;
    bp2 = (pos - TRIANGLE_BASEL/2) % NUM_POSITIONS;

    polar_point(bp1, TRIANGLE_BASER, &xb1, &yb1);
    polar_point(bp2, TRIANGLE_BASER, &xb2, &yb2);
    polar_point(pos, apex_radius, &xp, &yp);

    cons_xor_line(xb1, yb1, xb2, yb2, color);  /* base of triangle */
    cons_xor_line(xb1, yb1, xp, yp, color);  /* first arm */
    cons_xor_line(xb2, yb2, xp, yp, color);  /* second arm */
}
```

These graphic helper routines draw the line for the second hand and the triangle for the minute and hour hands. The **cons_xor_line** routine is similar to **cons_draw_line**, but exclusive-or's its pixels with the screen rather than just painting them. The xor technique is often used in graphics programming because it allows the drawing and erasing of objects without disturbing the background. When multiple objects overlap, the overlapping portions may become a strange color due to xor'ing, but you are guaranteed that when the objects are erased by xor'ing them a second time in the same location, whatever color was there before will be restored.

```
void check_for_quitkey(void)
{
    console_event_t ev;

    if(cons_poll_event(&ev) != 0 && ev.device == EVENT_KEYBOARD
        && ev.keycode == KEY_SPACE)
        quit();
}
```

The final helper routine polls the console keyboard for events, and quits the application if the space bar is pressed. The cons_poll_event|system call reports both key press and key release events, and reports a scan-code rather than an ASCII value. This interface is rather low level, but allows the application complete access to the up/down state of every key on the keyboard.

This completes the clock.mpl module. The clock.cfg config file follows:

```
#include "clock.h"

application clock {

    job  sec_job;      init sec_job: period SEC_PERIOD s;    /* jobs */
    job  min_job;      init min_job: period MIN_PERIOD s;
    job  hour_job;     init hour_job: period HOUR_PERIOD s;

    task ct;           start ct: clock;                     /* task */

    <> ct.sec_hand   <> in sec_job;                         /* threads */
    <> ct.min_hand   <> in min_job;
    <> ct.hour_hand  <> in hour_job;
}
```

Notice that the config file can include header files just like the MPL module can. This allows the programmer to put configuration-related constants in one header and use them in both the config file and the application modules.

The clock config simply creates one task, plus a job for each hand of the clock. The periods are defined in "clock.h":

```
#define INNER_RADIUS   235
#define OUTER_RADIUS   (INNER_RADIUS+15)
#define NUMBER_RADIUS  (OUTER_RADIUS+15)

#define TRIANGLE_BASER 30
```

```
#define TRIANGLE_BASEL 50

#define SEC_RADIUS    INNER_RADIUS

#define MIN_COLOR    YELLOW
#define MIN_RADIUS    (SEC_RADIUS-50)

#define HOUR_COLOR    GREEN
#define HOUR_RADIUS    (MIN_RADIUS-50)

#define NUM_POSITIONS  240
#define POS_PER_TICMARK (NUM_POSITIONS/60)

#define SEC_PERIOD    1                      /* jumps 1 tickmark/sec */
#define MIN_PERIOD    (60/POS_PER_TICMARK) /* creeps 1 tickmark/min */
#define HOUR_PERIOD (3600/5/POS_PER_TICMARK) /* creeps 5 tickmarks/hr */
```

First, a number of constants describing the visual appearance of the clock face are defined. These can be modified to taste.

Second, the timing characteristics of the program are given. The key parameter is NUM_POSITIONS, which gives the number of positions which the minute and hour hands take around the clock face. The larger this number, the smaller the distance the hands move each time, and the more frequently their jobs are executed. The minute hand must move through all 60 tick marks once every hour, and the hour hand 5 tick marks each hour. With NUM_POSITIONS set to 240, each hand moves four times for each tick mark on the face of the clock, which works out to one move every 15 seconds for the minute hand, and one move every 180 seconds for the hour hand.

Chapter 3

MPL / C References

Maruti Programming Language (MPL) is a simple extension to ANSI C to support modules, synchronization and communications primitives, and shared memory variables. MPL adds some restrictions that enable analysis of the CPU and memory requirements of the program. This chapter will define the MPL-specific features that differ from ANSI C.

3.1. EBNF Syntax Notation

In this manual, syntax is given in Extended Backus-Naur Formalism (EBNF). In this notation:

- literal strings are quoted, e.g. 'module'.
- other terminal symbols are bracketed, e.g. <module-name>.
- X|Y denotes alternatives.
- {Z} denotes zero-or-more.
- {X} denotes zero-or-one.

3.2. MPL Modules

The *module* is the compilation unit in MPL. It is presented to the MPL compiler as one file, but may contain normal C #include directives so that the parts of the module can be kept as distinct files. The MPL compiler generates a binary object file for the module, as well as a *partial EU graph file* for the module, which contains information about the module needed by the Maruti analysis tools.

At runtime, each MPL module is mapped to a Maruti *task*, which logically runs in its own address space. Communication between tasks is through *channels* or *shared blocks*. Each task can contain multiple *threads* of execution, each thread corresponding to an *entry* or *service* function of MPL.

Each module starts with the module name declaration:

module_name_spec ::= 'module' <module-name>.

3.3. Module Initialization

When the task corresponding to a module is loaded, the Maruti runtime system executes a non-real-time initializer function provided by the programmer. The initializer is a normal C function, but it must be present in every module. It is declared as:

int maruti_main(int argc, char **argv);

358

The job of this function is to initialize the state of the task, taking any parameter values into account. If the initializer returns 0, then the task is considered successfully loaded, otherwise the load fails. The initializer thread can not send or receive messages on Maruti channels.

3.4. Entry Functions

Maruti *entry* functions occur as top-level definitions in the MPL source file, similar in syntax to normal C function definitions.

> entry-function ::= 'entry' <entry-name> (' ')' entry-function-body.
> entry-function-body :: = channel-declaration-list c-function-body.

Entry functions serve as the top-level function of a Maruti thread which is invoked repeatedly with a period as specified externally, in the MCL configuration. Multiple instances of the entry thread can be active in a single task at runtime, so care must be taken to protect accesses to shared data with a region or local_region construct.

3.5. Service Functions

Maruti *service* functions also occur as top-level definitions in the MPL source file.

> service-function ::= 'service' <service-name>
>   '('<in-channel-name>':'<type_specifier>','<msg-ptr-name>')'
>   service-function-body.
> service-function-body :: = channel-declaration-list c-function-body.

Services are declared with the initiating channel and pointer to a message buffer. A service thread is invoked whenever a message on the channel has been received, thus it inherits the scheduling characteristics of the sender to the channel. Multiple instances of the service may be active in a single task at the same time, servicing messages from different senders, so care must be taken to protect accesses to shared data with a region or local_region construct.

The receipt of the invoking message into private storage is automatic, and the service function is called with a pointer to the message buffer. Fore example, given the service declaration:

> service consumer(inch: ch_type, msg) { ... }

The service is actually invoked as if it were a C function declared:

> void consumer(ch_type *msg) { ... }

359

3.6. MPL Channels

In Maruti, *channels* are one-way, typed, communications paths whose traffic patterns are analyzed and scheduled by the system. The channel end-points are declared as part of the *entry* or *service* functions which take part in the communication. The endpoints are connected in the MCL configuration for the application.

The syntax of MPL channels is similar to a C variable declaration:

channel-declaration-list ::= [channel-decl { channel-decl }].
    channel-declaration ::= channel-type channel { ',' channel } ';'.

    channel-type ::= 'out' | 'in' | 'in_first' | 'in_last'.
    channel ::= <channel-name> ':' type-specifier.

A channel endpoint declaration will normally be either an out endpoint or an in endpoint, used in the sending thread and receiving thread, respectively. There are two special variants of in endpoint, in_first and in_last, which denote asynchronous channels in which the communications will not be scheduled, and the input buffers are allowed to overflow. For in_first channels, the first messages received will be retained and the rest dropped, for in_last the most recent messages will be retained and older messages overwritten.

3.7. Communication Primitives

The message passing primitives appear as normal C function calls, but they are built in primitives of the MPL compiler, and their use is recorded so that the communications on the channel can be analyzed.

The three primitives each take a channel name and a pointer to a message buffer. Their declarations would look something like this:

```
void send    (out ch_name, ch_type* message_ptr);
void receive  (in ch_name, ch_type* message_ptr);
int optreceive(in ch_name, ch_type* message_ptr);
```

There are two variants of the receive primitive. A normal receive is used in most cases, and it raises an exception if there is no message delivered at the time it is executed. Normally the Maruti scheduler will arrange things so that this is never happens. When messages might not be present when the receiver is run, as when threads are communicating asynchronously with in_first and in_last channels, or when the sender sometimes will not send the message due to run time conditions, an optreceive must be used. The optreceive variant checks if a message is present, and receives it if so. It returns 1 if a message was delivered, or 0 if no message was delivered.

360

3.8. Critical Regions

Mutual exclusion is often necessary to prevent the corruption of data structures modified and accessed by concurrent threads. In Maruti, the region statement delineates a critical region.

region-statement ::= ('region'|'local_region') <region-name> c-statement.

The local region variant is used within a task, usually to serialize multiple thread access to data structures. The region variant is global to the application, and is used to serialize access to shared buffers and other application-defined resources, as specified in the MCL configuration for the application.

3.9. Shared Buffers

Finally, MPL adds shared buffers to the C language. Shared buffers declarations are similar in syntax to typedef declarations:

shared-buffer-decl ::= 'shared' <type-specifier> <shared-buffer-name>.

The shared buffer declaration is effectively a pointer declaration. For example:

shared some_type shared_buffer;

is treated as if it were a declaration of the form:

some_type *shared_buffer = &some_buffer;

The MPL specification for the application determines which tasks share each shared memory area. The runtime system takes care of allocating memory for the shared buffers, and initializing the buffer pointers. The MPL program can at all times dereference the pointer.

3.10. Restrictions to ANSI C in MPL

The Maruti real-time scheduling methodology requires that the tools be able to analyze the control flow and stack usage of the MPL programs, and that synchronization points be well known. Thus the following restrictions to ANSI C must be followed by the MPL programmer:

- No receive primitives are allowed within either loops or conditionals.
- No region construct are allowed within either loops or conditionals.
- No send primitive within a loop.
- Direct or indirect recursion is not allowed.
- Function calls via function pointers should not be used.

Chapter 4

MCL Reference

Maruti Configuration Language (MCL) is used to specify how individual program modules are to be connected together to form an application and to specify the details of the hardware platform on which the application is to be executed.

MCL is an interpreted C-like language. The MCL processor is called the *integrator*. The integrator interprets the instructions of the MCL program, instantiating and connecting the components of the application, checking for type correctness as it goes, and outputs the application graph and all allocation and scheduling constraints for further processing by other Maruti tools.

4.1. Top-level Declarations

Like a C program, an MCL configuration file is composed of a number of top-level declarations. The C preprocessor is invoked first, so the configuration file may contain #include and #define directives to make the configuration very customizable.

```
configuration ::= {toplevel-declaration}.
    toplevel-declaration ::= variable-declaration | system-declaration
                block-declaration | application-declaration.
```

The declarations may occur in any order---they do not have to be defined before used. The four types of top level declaration are described in more detail below.

4.1.1.   The Application Declaration

```
application-declaration ::= 'application' <application-name>
                '{' {instruction} '}'.
```

Like the main function of C, the application declaration is where the integrator will begin execution of the configuration directives. Only one application may be declared in the configuration.

4.1.2.   The System Declaration

```
system-declaration ::= 'system' <system-name>
                '{' {node-declaration} '}'.

node-declaration ::= 'node' <variable-name> ['with' attributes].
attributes ::= attribute {',' attribute}.

attribute ::= <symbol> ['=' <integer> | '=' <symbol> | '=' <string>].
```

Like the application declaration, the system declaration can occur at most once in a configuration. It is not needed for single-node operation. The system declaration names the nodes that an application will run on, and specifies attributes for them. For example:

362

```
system hdw {
    node northstar with address = "{0x00,0x60,0x8c,0xb1,0xfb,0xc6}", master;
    node raduga   with address = "{0x00,0x60,0x8c,0xb1,0xf6,0x67}";
}
```

The integrator does not assign any meaning to the attributes declared for the nodes, it just passes the information along. However, the Maruti binder does require the addresslattribute for each node, which specifies the node's ethernet address, and the master attribute on only one node, to specify which node will be the boot and time master. The Maruti/Virtual environment further requires that the node <variable-name> correspond to the hostname of the node in the testbed environment.

4.1.3.   Block Declarations

block-declaration ::= 'block' <block-name> '(' [block-parameters] ')'
                        block-parameter-channels
                        '{' {instruction} '}'.

A block is something like a function in C. When a block is declared, it may be called by any other block, except that no self-recursion is allowed. A block can not be declared inside another block. A block is called by giving its name and parameters. There are 2 kinds of parameters: classical parameters and channel parameters.

block-parameters ::= parameter { ',' parameter }.
parameter ::= ['var'] <parameter-name> ['[]'] [':' type].

Classical parameters are like function parameters in C or Pascal. They can be passed by value or by variable (var for variable passing). Arrays may also be be given as var parameters. The type of the parameter must be given for the first parameter. It may be omitted for following parameters: the integrator will assume that the parameter with no given type has the same type as the previous parameter.

block-parameter-channels ::= { ('in'|'out') channel-names ';' }.
channel-names ::= channel { ',' channel }.
channel ::= <channel-name> ['[' <integer> ']'].

The channel parameters describe the inputs and outputs of the block. The in and out keywords do not have exactly the same meaning has in MPL: they only show which channels are connected at the left and which are connected at the right of the block call (see connection below). The communication type of the channel in_first, in_last, or synchronous) and the type of the messages on the channel are determined by the connections of the channels to the tasks.

When there is an array of channel parameters, the connections will occur in ascending order. For example:

```
block foo()
    in ch[3];
    { ... }
```

```
application bar {
    channel a[3];
    <a[0..2]> foo() <>; /* a[0]->ch[0], a[1]->ch[2], a[2]->ch[2] */
    ...
}
```

4.1.4.   Variable Declarations

variable-declaration ::= type variable-names ';'.

type ::= 'float' | 'int' | 'string' | 'time' | 'channel'
         | 'task' | 'job' | 'node' | 'shared' | 'region'.

variable-names ::= variable { ',' variable }.
variable ::= <variable-name> ['[' <integer> ']'] ['[' <integer> ']'].

Variables may be declared globally at the top-level, or locally in a block. Global variables can be accessed in all blocks, while local variables can only be accessed in the block where they are declared. A local variable (or a parameter) may be declared with the same name as a global variable. In this case only the local variable (or the parameter) can be accessed in the block.
The order of the variable declarations does not matter. For example:

```
block foo()
{
    i = 4s + 5mn;      /* correct */
    time i;
}
```

Arrays may be declared. As in C, the array indicies are numbered from 0 to size-of-array less 1. Arrays of 1 or 2 dimensions are accepted. For example:

```
block foo()
{
    string s[10];
    s[5] = "a string";      /* correct */
    s[0] = s[5] + " foo";    /* correct */
    s[10] = "";              /* incorrect: out of array limits */
}
```

4.2. Instructions

The MCL integrator interprets a number of instructions that express the way an application is to be built up from components. The different instructions are explained below.

```
instruction ::= variable-declaration
            | task-initialization
            | job-initialization
            | connect-declaration
            | link-intruction
```

| allocation-instruction
| expression ';'
| print-instruction
| compound-instruction
| '{' {instruction} '}'.

4.2.1.   Compound Instructions

compound-instruction ::=
          'if' '(' test-expression ')' intruction
          | 'if' '(' test-expression ')' intruction 'else' instruction
          | 'do' instruction 'while' '(' test-expression ')' ';'
          | 'while' '(' test-expression ')' instruction
          | 'for' '(' expression ';' test-expression ';' expression ')'
          instruction.
   test-expression ::= expression.

The meaning of these constructs is the same as in the C language. The test-expression should evaluate to an integer, where 0 means false, and all other values mean true.

4.2.2.   Tasks

   task-initialization ::= 'start' names ':' <module-name> [module-parms]
              [instantiation] [task-allocation] ';'

   module-parms ::= '(' [module-parameter-list] ')'.
   module-parameter-list ::= expression {',' expression}.

   instantiation ::= 'with' <symbol> '=' constant
              {',' <symbol> '=' constant }.

   task-allocation ::= 'on' expression.

A variable of type task must be initialized before it can be used. This initialization consists of giving the name of a module: the task will be an instantiation of this module. Module parameters may be given: after evaluation they will be given to the initializer thread of the module. The initializations during the loading of an application will take place in exactly the same order as thay are found by the intergrator during the execution of the configuration.
      All the shared buffers and the global regions of the module must be instantiated using the with clause: the corresponding shared or region variables must be given.
      The on clause may be used to force allocation of the task on a particular node.

4.2.3.   Job Initialization

   job-initialization ::= 'init' names ':' timing-job ';'.
   timing-job ::= { 'period' expression }.

A variable of type job must be initialized before it can be used. The job will refer to a collection of threads with the same period.

4.2.4. Connections

connect-declaration ::= chan-list connect-name chan-list
                [in-job] {timing-service} [task-alloc]';'.
   chan-list ::= '<' [names] '>'.
   connect-name ::= <task-name> ['['expression']'] '.' <routine-name>
       | <block-name> '(' [expression {',' expression}] ')'.
   in-job ::= 'in' constant.
   timing-service ::= ('ready' expression | 'deadline' expression).

There are two types of connections: routine connections and block connections. In both cases the inputs are connected (or mapped) to the channels declared at the left of the connection and the outputs at the right. The number of input (or ouput) channels must be the same as in the definition of the routine (or the block). The mapping is done following the order of this definition.

     In a routine connection the inputs and ouputs of an entry or a service of a task are connected to channels. This connection creates a new instance of a service if the routine was a service, otherwise it creates the only instance of an entry. An entry can not be connected many times.

     For an entry connection a job name must be given, the entry will be a part of this job. For a service, a job can not be declared: the job of the service is implicitly given by the connection: the first input channel of a service is the triggering channel of the service. The job of the service is the same as the job of the origine of the triggering channel.

     A timing characterization may only be given to a routine connection.

     In a block connection the input and output channels of the block are mapped to the given channels. A mapping is also done for all the block parameters, following the order in the block definition. The number of parameters must be the same as in this definition, and all the types must be coherent.

4.2.5. Allocation Instructions

allocation-instruction ::= 'separate' '(' names ')' ';'
                   | 'together' '(' names ')' ';'.

A separate instruction is a command to the allocator to keep the tasks on different nodes in the final system. A together instruction specifies that all tasks must be allocated to the samenode.

4.2.6. Link Instruction

   link-intruction ::= 'link' expression 'to' expression ';'.

In a few cases the connections are not sufficient to describe a communication graph with the structure of the blocks. In these cases a link instruction may be used.

     A link between two channels means that the two channels are the same.

     Example: if we want to connect directly an input and an output channel of a block a link must be used.

```
block foo()
  in in_channel;
  out out_channel;

  {
    link in_channel to out_channel;
  }
```

4.2.7.   Print Instruction

print-instruction ::= 'print (' expression {',' expression} ');'.

The print instruction outputs messages to the standard output during integration. This instruction can be used for the debugging of a configuration file. Any string, number, or time may be printed. A newline is added at the end.

4.3. Expressions

Expressions in MCL are very similar to C expressions:

```
expression ::= expression '=' expression
| expression '||' expression
| expression '&&' expression
| expression '==' expression
| expression '!=' expression
| expression '<' expression
| expression '>' expression
| expression '<=' expression
| expression '>=' expression
| expression  ('d'|'h'|'mn'|'s'|'ms'|'us')
| expression '+' expression
| expression '-' expression
| expression '*' expression
| expression '/' expression
| expression '%' expression
| '!' expression
| '(' expression ')'
| expression '++'
| expression '--'
| constant

constant ::= <symbol> [ '[' expression ']' ] [ '[' expression ']' ]
| <symbol> '[' expression '..' expression ']' [ '[' expression ']' ]
| <symbol> '[' expression ']' '[' expression '..' expression ']'
| <integer>
| <double>
| <string>.
```

In addition to the usual C expressions, MCL supports time unit expressions, for example, '3 s + 500 ms' is a time expression that evaluates to 3.5 seconds.

Also, MCL supports array range notation as a shorthand for lists. For example, the expression 'c[2..4]' is shorthand for c[2], c[3], c[4]'. This notation is most often used for passing arrays of channel values to blocks or in connection instructions.

CHAPTER 5

Maruti Runtime System Reference

The Maruti runtime system is bound together with the application binary files by the *mbind* utility. Only those parts of the runtime needed by the application are linked in. There are several versions of the runtime system available depending on the environment in which the application will be run. For example, there are two different versions of the core library: a stand-alone version that can boot directly on bare hardware, and a Unix version that runs as a user-level process under Unix, providing virtual-time execution and access to debugging tools.

The set of library versions that an application links with are called *flavors*. Flavors are specified by the programmer as strings of library names separated by a '+', for example, 'ux+x11'.

5.1. Core Library Reference

#include <maruti/maruti-core.h>

The Maruti core library implements the scheduling, thread and memory management, and network communication subsystem. It provides primitives for applications to send and receive messages, insert preemption points, manipulate the schedule (via calendars), and do time and date calculations. There are currently two flavors of the core library:

- **sa** -- The Maruti/Standalone core library. Applications linked with this flavor can be booted directly (by the NetBSD boot blocks). It includes the distributed operation support, based on the 3Com 3c507 Etherlink/16 adapter.

- **ux** -- The Maruti/Virtual debugging core library. Applications linked with this flavor are run as normal Unix processes from the NetBSD command line. It includes a virtual-time scheduler and debugging monitor (described below) and implements distributed operation using normal Unix TCP/IP networking facilities.

5.1.1.　MPL　Built-in　Primitives

void maruti_eu(void)

The maruti_eu primitive inserts a Maruti EU break into the program at the location of the call. It is not normally used explicitly in an application, as the system tools put EU breaks where necessary for synchronization. It is useful, however, for breaking up long-running EUs -- the maruti_eu then serves as a possible preemption point.

```
void send(out ch_name, ch_type* message_ptr)
void receive(in ch_name, ch_type* message_ptr)
int optreceive(in ch_name, ch_type* message_ptr)
```

369

The communications primitives are documented in section 3.7. in the MCL Reference Chapter.

5.1.2. Calendar Switching

int maruti_calendar_activate(int calendar_num, mtime switch_time, mtime offset_time)
void maruti_calendar_deactivate(int calendar_num, mtime switch_time)

- Maruti calendars may be activated and deactivated (*switched* on or off) at any time. The switch_time is the time at which the de/activation should take place. The switch can occur at any point in the future, and the switch requests can come out of order with respect to the switch time. Requests with the same switch time are executed in the order of the requests.

Calendars can be activated with a particular offset_time, which is the relative position within the calendar to start executing at the switch time. The offset time will normally be zero, but can be any relative time up to the lcm time of the calendar.

The runtime system does not check the feasibility of the combined schedules represented by the calendars - that should be done offline.

5.1.3. Calendar Modification

void maruti_calendar_create(int calendar_num, int num_entries, mtime lcm_time) void maruti_calendar_delete(int calendar_num)

Calendars are normally created offline and compiled into the Maruti application, but it is possible to create new calendars at runtime. The application is responsible for insuring that the generated schedules are feasible.

When a calendar is created, the maximum number of entries it will contain must be specified, as well as the lcm_time, which is the period of the calendar as a whole. At the end of its period, the calendar will wrap around and begin executing from the beginning again.

```
typedef struct calendar_s {
    entry_t *entries;
    int num_entries;
    mtime lcm_time;
    mtime base_time;
    entry_t *cur;        /* cur-entries is the current offset */
    <...>
} calendar_t;
typedef struct {
    int eu_thread, eu_id;
    mtime eu_start, eu_deadline;
    int eu_type;
#    define EU_EMPTY     0    /* empty EU slot */
#    define EU_PERIODIC  1    /* periodic EU */
    <...>
} entry_t;
```

370

```
void maruti_calendar_get_header(int calendar_num, calendar_t *calendarp)
void maruti_calendar_get_entry(int calendar_num, int entry_num, entry_t *entryp)
void maruti_calendar_set_entry(int calendar_num, int entry_num, entry_t entry)
```

The maruti_calendar_set_entry call is used to populate new calendars. It can overwrite any entry in any inactive calendar. The entry eu_start and eu_deadline times are the earliest start time and latest end time, respectively. The eu_id serves to identify the eu when tracing or reporting timing results.

. The maruti_calendar_get_header and maruti_calendar_get_entry calls can be used to query the contents of a calendar. These are useful when 'cloning' an existing calendar into a new calendar, perhaps with modifications.

5.1.4. Date and Time Manipulation

```
#include <maruti/mtime.h>
```

The Maruti core library provides routines and macros for simple time and date calculations.

```
    typedef struct {
        long seconds;
        long microseconds;
    } mtime;
```

```
#define time_cmp(a,b)         /* like strcmp, 0 if eq, lt 0 if a < b, etc */
#define time_add(a,b)       .  /* a += b */
#define time_sub(a,b)         /* a -= b */
#define time_add_scalar(t, s)  /* t += s (s is an int, in microseconds) */
#define time_sub_scalar(t, s)  /* t -= s (s is an int, in microseconds) */
#define time_mul_scalar(t, s)  /* t *= s (s is an int) */
#define time_div_scalar(t, s)  /* t /= s (s is an int) */
```

The mtime type is the basic Maruti time structure. A number of convenience macros for arithmetic on mtime values are provided. Two mtime values may be compared, added, or subtracted. In addition, an integer time in microseconds may be added to and subtracted from an mtime value, and mtime values may be multiplied or divided by integer scaling factors.

Note: The microseconds field is always in the range 0 to 999999, and the time represented by an mtime value is always the number of seconds plus the number of microseconds. These rules hold even for negative mtime values, which can arise when subtracting mtimes. Thus the mtime representation for the time -1.3 seconds is { -2, 700000 }.

```
    void maruti_get_current_time(mtime *curtime)
```

The current system time is returned by maruti_get_current_time. Maruti, like Unix, represents absolute time as the number of seconds and microseconds since the *Epoch* time, defined as 00:00 GMT on January 1, 1970.

371

```
typedef struct {
    short year;              /* year - 1900 */
    short month;             /* month (0..11) */
    short wday;              /* day of week  (0..6) */
    short mday;              /* day of month (1..31) */
    short yday;              /* day of year (0..365) */
    short second, minute;    /* 0..59 */
    short hour;              /* 0..23 */
    int microsecond;         /* 0..999999 */
} mdate;
```

```
mtime maruti_date_to_time(mdate d)
mdate maruti_time_to_date(mtime t)
```

```
mtime maruti_gmtdate_to_time(mdate d)
mdate maruti_time_to_gmtdate(mtime t)
```

```
int maruti_set_gmtoff(int gmtoff)
int maruti_get_gmtoff(int *gmtoffp)
```

Applications will often want to view the time as something more convenient than the number of seconds since the Epoch. The Maruti mdate type denotes a time expressed as a date plus a time of day. The functions maruti_time_to_gmtdate and maruti_gmtdate_to_time convert between mtime and mdate values using the GMT timezone. The functions maruti_time_to_date and maruti_date_to_time convert using the local offset from GMT.

The local timezone used in these conversions is initially set by the runtime system, but may be changed by the application. The timezone is expressed as an offset from GMT in seconds. For example the U.S. timezone EST is 5 hours behind GMT, or -18000 seconds offset.

Note: Maruti does not at this time attempt to handle leap seconds or automatically switching the local timezone to account for daylight savings times. The cost of providing these features in code and table space was deemed prohibitive.

5.1.5. Miscellaneous Functions

```
void quit(int exitcode)
```

The quit call terminates the application. The exit code is not usually relevant in an embedded system, but will be returned to the environment where that makes sense (such as in the Unix debugging environment).

5.2. Console Library Reference

#include <maruti/console.h>

The Maruti console graphics library provides access to the console device, including the keyboard and speaker, but most importantly the graphical display. The graphics library includes support for placing text anywhere on the screen, simple 2d geometry primitives suitable for generating line and bar graphs, and includes optimized routines for moving bitmaps without flicker, for animated simulations. There are currently three flavors of the graphics library implemented:

- **et4k** -- This flavor supports Super VGA graphics cards based on the Tseng Labs ET4000 chip and its accelerated descendents, like the ET4000/W32. The et4k flavor runs the screen at a resolution of 1024x768 in 256 color mode.

- **vga16** -- This flavor supports all standard VGA graphics cards, running the screen at a resolution of 640x480, in 16 color banked mode.

- **x11** -- This flavor works with the {\bf ux} core flavor, displaying the Maruti screen in an X11 window under Unix.

5.2.1. Screen Colors

The Maruti console graphics library supports the following colors, defined in <maruti/console.h>:

```
#define BLACK           0
#define DARK_BLUE       1
#define DARK_GREEN      2
#define DARK_CYAN       3
#define DARK_RED        4
#define DARK_VIOLET     5
#define DARK_YELLOW     6
#define DARK_WHITE      7
#define BROWN           8
#define BLUE            9
#define GREEN           10
#define CYAN            11
#define RED             12
#define VIOLET          13
#define YELLOW          14
#define WHITE           15
/* aliases */
#define GREY            DARK_WHITE
#define GRAY            DARK_WHITE
```

The maximum screen size supported is also defined:

```
#define CONSOLE_WIDTH       1024
#define CONSOLE_HEIGHT      768
```

5.2.2.   Graphics Functions

void cons_graphics_init(void)

The cons_graphics_init function must be called before any other graphics functions, usually from the maruti_main function of the application's screen driver task.

void cons_fill_area(int x, int y, int width, int height, int color)
void cons_xor_area(int x, int y, int width, int height, int color)

These functions paint an area of the screen, specified by its upper-left coordinates (x, y), and its width and height, in the given color. The cons_fill_area variant overwrites the previous contents of that area of the screen,while cons_xor_area exclusive-or's the screen contents with the specified color.

function cons_draw_pixel(int x, int y, int color)
function cons_xor_pixel(int x, int y, int color)

These functions draw and xor, respectively, a single pixel at (x, y) in the specified color.

void cons_draw_line(int x1, int y1, int x2, int y2, int color)
void cons_xor_line(int x1, int y1, int x2, int y2, int color)

These functions draw and xor, respectively, a single-pixel width line from coordinates (x1, y1) to (x2, y2) in the specified color.

void cons_draw_bitmap(int x, int y, int width, int height,
                void *bitmap, int color)
void cons_xor_bitmap(int x, int y, int width, int height,
                void *bitmap, int color)

These functions draw and xor, respectively, a width-by-height sized bitmap onto the screen in the specified color, with its upper-left corner at (x, y). The bitmap is in standard X bitmap format, with eight pixels per byte, and an even multiple of eight pixels per scan line.

void cons_move_bitmap(int x1, int y1, int x2, int y2, int width, int height,

```
                    void *bitmap, int color)
void cons_xor_move_bitmap(int x1, int y1, int x2, int y2, int width, int height,
                    void *bitmap, int color)
```

These functions optimize the erasing and redrawing of a bitmap by combining the operations into one loop, modifying one scan-line at a time. This optimization eliminates the flicker that can occur when erasing the entire bitmap then redrawing it, making animations more effective.

The call cons_move_bitmap(x1,y1,x2,y2,w,h,b,c) is equivalent to the sequence:

```
        cons_draw_bitmap(x1,x2,w,h,b,BLACK);
        cons_draw_bitmap(x2,y2,w,h,b,c);
```

The call cons_xor_move_bitmap(x1,y1,x2,y2,w,h,b,c) is equivalent to the sequence:

```
        cons_xor_bitmap(x1,y1,w,h,b,c);
        cons_xor_bitmap(x2,y2,w,h,b,c);


        void cons_puts(int x, int y, int color, char *string)
        void cons_xor_puts(int x, int y, int color, char *string)
```

These functions draw and xor, respectively, a text string at (x, y) in the specified color.

5.2.3.   Keyboard and Speaker functions

```
        typedef struct
        {
          unsigned char device;     /* just keyboard works for now */
        #     define EVENT_OTHER        0
        #     define EVENT_KEYBOARD     2
          unsigned char keycode;
        } console_event_t;


    int cons_poll_event(console_event_t *event)
```

The cons_poll_event call returns 1 if a console event has occurred, 0 otherwise. There there is a pending console event, the event structure is filled in. The device field is set to EVENT_KEYBOARD and the keycode field is set to the scan code of the key that was pressed. The list of scan codes is in <maruti/keycodes.h>.

```
        void cons_start_beep(int pitch)
         void cons_stop_beep(void)
```

The console speaker can be turned on and off with these functions. The cons_start_beep call programs the speaker to sound at a particular frequency, in hertz, and cons_stop_beep turns it

off.

5.3. Maruti/Virtual Monitor

The *ux* flavor of the Maruti core library includes some basic debugging facilities called the Maruti *monitor*. While an application compiled with *ux* is running, aspects of its execution can be controlled from the Unix tty (which will be distinct from the console keyboard device). The monitor provides the following facilities:

Tracing scheduler actions. The user can independently toggle the tracing of elemental unit executions, calendar wrap-around events, and calendar-switch events.

Single-stepping calendars or elemental units. The user can toggle single stepping through each elemental unit execution, or a whole calendar's execution.

Controlling virtual-time execution speed. The user can control the speed of the application in two ways. First, the user can toggle as-soon-as-possible execution of elemental units, called *asap mode*. Second, the user can set the speed at which virtual time advances relative to real clock time.

Both single-keystroke and command-line operation. All monitor switches may be toggled with a single keystroke while the application continues running. Also, the user can enter a command-line mode in which various parts of the system state may be queried and modified.

5.3.1.   Controlling Virtual Time

The Maruti monitor contains a user-settable *speed* variable which determines the rate at which virtual time advances relative to the actual clock time.

The speed may be set to any floating point value greater than zero. Thus virtual speed may be set to run, for example, five times faster than clock time (speed = 5) or at four times slower (speed = 0.25). The speed is logically limited on the side by the utilization of the CPU. The execution of application code can not be sped up, only the idle time between executions.

Idle time can be eliminated completely by turning on as-soon-as-possible scheduling of elemental unit (*asap-mode*). In asap-mode the virtual time is advanced to the start time of the next elemental unit as soon as the previous one completes, resulting in the execution of all EUs in immediate succession. Asap-mode is separate from the *speed* variable -- it can be toggled independently, and when turned off, scheduling continues at the previously set speed.

5.3.2.   Single-Keystroke Operation

The following keys are active from the Unix tty session (**not** the console keyboard) while the application is running:

? shows the list of keystrokes and current values for the toggle switches.

a toggle as-soon-as-possible mode.

e toggle elemental unit tracing.

c toggle calendar tracing.

x toggle calendar-switch tracing.

s toggle elemental unit single-stepping.

S. toggle calendar single-stepping.

q quit application completely.

<ESC> stop application and enter command-line mode.

5.3.3. Command-line Operation

The following commands are available from command line mode. At this time, command-line mode is a just a framework with just a few commands. More commands to query and set the system state are envisioned for future releases.

help
Get a list of command-line mode commands.

quit
Quit the application completely.

vars
Show all user-settable monitor variables and their values.

speed <value>
Set the virtual-time speed to **value.** The value can be any floating point value greater than zero.

cstep [on|off]
Set or toggle calendar single-stepping.

estep [on|off]
Set or toggle eu single-stepping.

ctrace [on|off]
Set or toggle calendar tracing.

etrace [on|off]
Set or toggle eu tracing.

strace [on|off]
Set or toggle calendar switch tracing.

Chapter 6

Maruti Tool Reference

6.1. Maruti Builder

The mbuild program automates the process of building a runnable Maruti application. This involves building the constituent application binaries, integrating and scheduling the application, and binding the application with the desired Maruti runtime flavor.

    Mbuild is normally run in the directory in which the application config file and constituent module source files are located. It will automatically find the config file by its .cfg extension, read it, and generate a makefile that builds what modules it finds used there, then calls the other Maruti tools. Mbuild works by creating an obj subdirectory, and putting all output files there.

    If there is more than one config file in the current directory, the desired file must be specified with the -f <config file> option.

    The user may optionally customize the mbuild actions by providing an Mbuild.inc file in the current directory. This file will be included into the makefile generated by mbuild. In addition to providing additional build targets and dependency lines, the user may set some variables to modify the mbuild actions themselves:

FLAVORS Default: **ux+x11 ux+et4k sa+et4k** The list of runtime flavors with which to link the application.

MPC Default: mpc. The program executed to compile MPL programs. Not normally modified by users.

MPC_FLAGS Default: <empty>. Supplemental flags for the MPL compiler. Most GCC flags will work here. Most often the user will want to customize the include directories with -I <dir>.

CFG Default: cfg. The program executed to interpret the MCL config file and integrate the application. Not normally modified by users.

CFG_FLAGS Default: <empty>. Supplemental flags for the MPC integrator. Not normally modified by users.

ALLOCATOR Default: allocator. The program executed to allocate and schedule the application. Not normally modified by users.

ALLOCATOR_FLAGS Default: -p 1. Flags for the Allocator. See section 6.4. on the Allocator below for more details.

MBIND Default: mbind. The program executed for binding the application and runtime system. Not normally modified by the users.

MBIND_FLAGS Default:<empty>. Flags for the Mbind program. Not normally modified by users.

6.2. MPL/C Compiler

The MPL/C compiler (mpc) consists of a modified gcc plus some attendant scripts to post-process the compiler output. It generates a .o file for a module, plus a .eul file containing a partial elemental-unit graph to be read by the integrator.

The mpc program will accept GCC command-line options. See the gcc(1) manual page for details on the available options. The most commonly used option will be -I dir to customize the include directories.

6.3. MPL/C Integrator

The MCL Integrator (cfg) reads the application config file (*appname.cfg*) and all the module elemental-unit graph files (*modulename.eul*), then generates and checks all the jobs, tasks, threads, and connections for the application. It outputs a loader map file (*appname.ldf}*), and a complete application elemntal-unit graph annotated with allocation and scheduling constraints and communication parameters (*appname.sch*). There are no cfg options normally used.

6.4. Allocator/Scheduler

The Allocator/Scheduler (**allocator**) attempts to find a valid allocation for the application tasks across the nodes of the network, and a valid schedule for each node and for the network bus. The allocation and schedules are considered valid if all allocation, communication, and scheduling constraints are met.

The allocator/scheduler stops when a valid allocation and schedule is found, or when it is determined that one cannot be found. There is no attempt to load-balance the nodes or minimize network communications beyond what is needed for a minimally valid schedule. The allocator outputs an allocation information file (*appname.alloc*) and calendar schedules file (*appname.cal*).
The allocator takes two flags:

-p <number of processors> Default: 1. The number of processors in the target system. It should match the number of nodes defined in the config file.

-t <tdma slot size> Default: 1000. The Time Division Multiplexed Access (TDMA) slot size for the network bus. This is the time, in microseconds, that each node will be alloted to transmit on the network. All the nodes get a TDMA slot in turn. The tdma slot size should stay between 1000 and 16000 microseconds, depending on the application's latency requirements and the network hardware's buffering capacity.

6.5. Maruti Binder

The Maruti Binder (mbind) reads in the loader map .ldf file, the allocation .alloc file, and the calendars .cal file, and generates the static data structures needed by the runtime system (*appname-globals.c*). It also generates a makefile (*appname-bind.mk*) that manages the linking of each task of the application within its own logical address space, then linking all tasks together with the various flavors of the runtime library.

6.6. Timing Trace Analyzer

The Timing Trace Analyser (timestat) takes a list of timing output files as generated by the runtime system and generates a .wcet file that contains the worst case execution times for the elemental units, as needed by the allocator. Timestat also prints other statistics generated by the runtime system.

6.7. Timing Stats Monitor

Timing information is output from a stand-alone Maruti system through a serial port when the application terminates. The mgettimes program, running on another computer connected to the other end of that serial line, will receive the timing data and store it in a file suitable for processing by timestat. Mgettimes can process the output of multiple runs on the test setup, even from different applications. Simply leave the program running and any data that is received will be saved.

Mgettimes is called as follows:

mgettimes <speed> <serial-port>

where <speed> is the communications rate at which the times will be output (19200 bps in the default core), and <serial-port> is the device file for the communications port (for example, /dev/tty00 for the PC's COM1 port).

Maruti 3.1
Design Overview
First Edition

Systems Design and Analysis Group
Department of Computer Science
University of Maryland at College Park

1. Introduction

Many complex, mission critical systems depend not only on correct functional behavior, but also on correct temporal behavior. These systems are called. The most critical systems in this domain are those which must support applications with hard real-time constraints, in which missing a deadline may cause a fatal error. Due to their criticality, jobs with hard real-time constraints must always execute satisfying the user specified timing constraints, despite the presence of faults such as site crashes or link failures.

A real-time operating system, besides having to support most functions of a conventional operating system, carries the extra burden of guaranteeing that the execution of its requested jobs will satisfy their timing constraints. In order to carry out real-time processing, the requirements of the jobs have to be specified to the system, so that a suitable schedule can be made for the job execution. Thus, conventional application development techniques must be enhanced to incorporate support for specification of timing and resource requirements. Further, tools must be made available to extract these requirements from the application programs, and analyze them for schedulability.

Based on the characteristics of its jobs, a real-time system can be classified as static, dynamic or. In a static system, all (hard real-time) jobs and their execution characteristics are known ahead of time, and thus can be statically analyzed prior to system operation. Many such systems are built using the cyclic executive or static priority architecture. In contrast, there are many systems in which new processing requests may be made while the system is in operation. In a dynamic system, new requests arrive asynchronously and must be processed immediately. However, since new requests demand immediate attention, such systems must either have "soft" constraints, or be lightly loaded and rely on exception mechanisms for violation of timing constraints. In contrast, reactive systems have certain lead time to decide whether or not to accept a newly arriving processing request. Due the presence of the lead time, a reactive system can carry out analysis without adversely affecting the schedulability of currently accepted requests. If adequate resources are available then the job is accepted for execution. On the other hand, if adequate resources are not available then the job is rejected and does not execute. The ability to reject new jobs distinguishes a reactive system from a completely dynamic system.

The purpose of the Maruti project is to create an environment for the development and deployment of critical applications with hard real-time constraints in a reactive environment. Such applications must be able to execute on a platform consisting of distributed and heterogeneous resources, and operate continuously in the presence of faults.

The Maruti project started in 1988. The first version of the system was designed as an object-oriented system with suitable extensions for objects to support real-time operation. The proof-of-concept version of this design was implemented to run on top of the Unix operating system and supported hard and non-real-time applications running in a

distributed, heterogeneous environment. The feasibility of the fault tolerance concepts incorporated in the design of Maruti system were also demonstrated. No changes to the Unix kernel were made in that implementation, which was operational in 1990. We realized that Unix is not a very hospitable host for real-time applications, as very little control over the use of resources can be exercised in that system without extensive modifications to the kernel. Therefore, based on the lessons learned from the first design, we proceeded with the design of the current version of Maruti and changed the implementation base to CMU Mach which permitted a more direct control of resources.

Most recently, we have implemented Maruti directly on 486 PC hardware, providing Maruti applications total control over resources. The initial version of the distributed Maruti has also been implemented, allowing Maruti applications to run across a network in a synchronized, hard real-time manner.

In this paper, we summarize the design philosophy of the Maruti system and discuss the design and implementation of Maruti. We also present the development tools and operating system support for mission critical applications. While the system is being designed to provide integrated support for multiple requirements of mission critical applications, we focus our attention on real-time requirements on a single processor system.

2. Maruti Design Goals

The design of a real-time system must take into consideration the primary characteristics of the applications which are to be supported. The design of Maruti has been guided by the following application characteristics and requirements.

- **Real-Time Requirements.** The most important requirement for real-time systems is the capability to support the timely execution of applications. In contrast with many existing systems, the next-generation systems will require support for hard, soft,and non-real-time applications on the same platform.

- **Fault Tolerance.** Many mission-critical systems are safety-critical, and therefore have fault tolerance requirements. In this context, fault tolerance is the ability of a system to support continuous operation in the presence of faults.

Although a number of techniques for supporting fault-tolerant systems have been suggested in the literature, they rarely consider the real-time requirements of the system. A real-time operating system must provide support for fault tolerance and exception handling capabilities for increased reliability while continuing to satisfy the timing requirements.

- **Distributivity.** The inherent characteristics of many systems require that multiple autonomous computers, connected through a local area network, cooperate in a distributed manner. The computers and other resources in the system may be homogeneous or heterogeneous. Due to the autonomous operation of the components which cooperate, system control and coordination becomes a much more difficult task than if the system were implemented in a centralized manner. The techniques learned in the design and implementation of centralized systems do not always extend to distributed systems in a straightforward manner.

- **Scenarios.** Many real-time applications undergo different modes of operation during their life cycle. A scenario defines the set of jobs executing in the system at any given time. A hard real-time system must be capable of switching from one scenario to another, maintaining the system in a safe and stable state at all times, without violating the timing constraints.

- **Integration of Multiple Requirements.** The major challenge in building operating systems for mission critical computing is the integration of multiple requirements. Because of the conflicting nature of some of the requirements and the solutions developed to date, integration of all the requirements in a single system is a formidable task. For example, the real-time requirements preclude the use of many of the fault-handling techniques used in other fault-tolerant systems.

3. Design Approach and Principles

Maruti is a time-based system in which the resources are reserved prior to execution. Resource reservation is done on the time-line, thus allowing for reasoning about real-time properties in a natural way. The time-driven architecture provides predictable execution for real-time systems, a necessary requirement for critical applications requiring hard real-time performance. The basic design approach is outlined below:

- **Resource Reservation for Hard Real-Time Jobs.** Hard real-time applications in Maruti have advance resource reservation resulting in a priori guarantees about the timely execution of hard real-time jobs. This is achieved through a calendar data structure which keeps track of all resource reservations and the assigned time intervals. The resource requirements are specified as early as possible in the development stage of an application and are manipulated, analyzed, and refined through all phases of application development.

- **Predictability through Reduction of Resource Contention.** Hard real-time jobs are scheduled using a time-driven scheduling paradigm in which the resource contention between jobs is eliminated through scheduling. This results in reduced runtime overheads and leads to a high degree of predictability. However, not all jobs can be pre-scheduled. Since resources may be shared between jobs in the calendar and other jobs in the system, such as non-real-time activities, there may be resource contention leading to lack of predictability. This is countered by eliminating as much resource contention as possible and reducing it whenever it is not possible to eliminate it entirely. The lack of predictability is compensated for by allowing enough slack in the schedule.

- **Integrated Support for Fault Tolerance.** Fault tolerance objectives are achieved by integrating the support for fault tolerance at all levels in the system design. Fault tolerance is supported by early fault detection and handling, resilient application structures through redundancy, and the capability to switch modes of operation. Fault detection capabilities are integrated into the application during its development, permitting the use of application specific fault detection and fault handling. As fault handling may result in violation of temporal constraints, replication is used to make the application resilient. Failure of a replica may not affect the timely execution of other replicas and, thereby, the operation of the system it may be controlling. Under anticipated load and failure conditions, it may become necessary for the system to revoke the guarantees given to the hard real-time applications and change its mode of operation dynamically so that an acceptable degraded mode of operation may continue.

- **Separation of Mechanism and Policy.** In the design of Maruti, an emphasis has been placed on separating mechanism from policy. Thus, for instance, the system provides basic dispatching mechanisms for a time-driven system, keeping the design of specific scheduling policies separate. The same approach is followed in other aspects of the system. By separating the mechanism from the policy, the system can be tailored and optimized to different environments.

- **Portability and Extensibility.** Unlike many other real-time systems, the aim of the Maruti project has been to develop a system which can be tailored to use in a wide variety of situations---from small embedded systems to complex mission-critical systems. With the rapid change in hardware technology, it is imperative that the design be such that it is portable to different platforms and makes minimal assumptions about the underlying hardware platform. Portability and extensibility is also enhanced by using modular design with well defined interfaces. This allows for integration of new techniques into the design with relative ease.

- **Support of Hard, Soft, and Non-Real-Time in the Same Environment.** Many critical systems consist of applications with a mix of hard, soft, and non-real-time requirements. Since they may be sharing data and resources, they must execute within the same environment. The approach taken in Maruti is to support the integrated execution of applications with multiple requirements by reducing and bounding the unpredictable interaction between them.

- **Support for Distributed Operation.** Many embedded systems require several processors. When multiple processors function autonomously, their use in hard real-time applications requires operating system support for coordinated resource management. Maruti provides coordinated, time-based resource management of all resources in a distributed environment including the processors and the communication channels.

- **Support for Multiple Execution Environments.** Maruti provides support for multiple execution environments to facilitate program development as well as execution. Real-time applications may execute in the Maruti/Mach or Maruti/Standalone environments and maintain a high degree of temporal determinacy. The Maruti/Standalone environment is best suited for the embedded applications while Maruti/Mach permits the concurrent execution of hard real-time and non-real-time Unix applications. In addition, the Maruti/Virtual environment has been designed to aid the development of real-time applications. In this environment the same code which runs in the other two environments can execute while access to all Unix debugging tools is available. In this environment, temporal accuracy is maintained with respect to a virtual real-time.

- **Support for Temporal Debugging.** When an application executes in the Maruti/Virtual environment its interactions are carried out with respect to virtual real-time which is under the control of the user. The user may speed it up with respect to actual time or slow it down. The virtual time may be paused at any instant and the debugging tools used to examine the state of the execution. In this way we may debug an application while maintaining all temporal relationships, a process we call temporal debugging.

4. Application Development Environment

To support applications in a real-time system, conventional application development techniques and tools must be augmented with support for specification and extraction of resource requirements and timing constraints. The application development system provides a set of programming tools to support and facilitate the development of real-time applications with diverse requirements. The Maruti Programming Language (MPL) is used to develop individual program modules. The Maruti Configuration Language (MCL) is used to specify how individual program modules are to be connected together to form an application and the details of the hardware platform on which the application is to be executed.

4.1. Maruti Programming Language

Rather than develop completely new programming languages, we have taken the approach of using existing languages as base programming languages and augmenting them with Maruti primitives needed to provide real-time support.

In the current version, the base programming language used is ANSI C. MPL adds *modules, shared memory blocks, critical regions, typed message passing, periodic functions*, and *message-invoked functions* to the C language. To make analyzing the resource usage of programs feasible, certain C idioms are not allowed in MPL; in particular, recursive function calls are not allowed nor are unbounded loops containing externally visible events, such as message passing and critical region transitions.

- The code of an application is divided into *modules*. A module is a collection of procedures, functions, and local data structures. A module forms an independently compiled unit and may be connected with other modules to form a complete application. Each module may have an *initialization function* which is invoked to initialize the module when it is loaded into memory. The initialization function may be called with arguments.

- Communication primitives send and receive messages on one-way typed *channels*. There are several options for defining channel endpoints that specify what to do on buffer overflow or when no message is in the channel. The connection of two endpoints is done in the MCL specification for the application-Maruti insures that endpoints are of the same type and are connected properly at runtime.

- Periodic functions define *entry points* for execution in the application. The MCL specification for the application will determine when these functions execute.

- Message-invoked functions, called *services*, are executed whenever messages are received on a channel.

- *Shared memory blocks* can be declared inside modules and are connected together as specified in the MCL specifications for the application.

- An *action* defines a sequence of code that denotes an externally observable action of the module. Actions are used to specify timing constraints in the MCL specification.

385

- *Critical Regions* are used to safely access and maintain data consistency between executing entities. Maruti ensures that no two entities are scheduled to execute inside their critical regions at the same time.

4.2. Maruti Configuration Language

MPL Modules are brought together into as an executable application by a specification file written in the Maruti Configuration Language (MCL). The MCL specification determines the application's hard real-time constraints, the allocation of tasks, threads, and shared memory blocks, and all message-passing connections. MCL is an interpreted C-like language rather than a declarative language, allowing the instantiation of complicated subsystems using loops and subroutines in the specification. The key features of MCL include:

- **Tasks, Threads, and Channel Binding.** Each module may be instantiated any number of times to generate tasks. The threads of a task are created by instantiating the entries and services of the corresponding module. An entry instantiation also indicates the job to which the entry belongs. A service instantiation belongs to the job of its client. The instantiation of a service or entry requires binding the input and output ports to a channel. A channel has a single input port indicating the sender and one or more output ports indicating the receivers. The configuration language uses channel variables for defining the channels. The definition of a channel also includes the type of communication it supports, i.e., synchronous or asynchronous.

- **Resources.** All global resources (i.e., resources which are visible outside a module) are specified in the configuration file, along with the access restrictions on the resource. The configuration language allows for binding of resources in a module to the global resources. Any resources use by a module which are not mapped to a global resource are considered local to the module.

- **Timing Requirements and Constraints.** These are used to specify the temporal requirements and constraints of the program. An application consists of a set of cooperating jobs. A job is a set of entries (and the services called by the entries) which closely cooperate. Associated with each job are its invocation characteristics, i.e., whether it is periodic or aperiodic. For a periodic job, its period and, optionally, the ready time and deadline within the period are specified. The constraints of a job apply to all component threads. In addition to constraints on jobs and threads, finer level timing constraints may be specified on the observable actions. An observable action may be specified in the code of the program. For any observable action, a ready time and a deadline may be specified. These are relative to the job arrival. An action may not start executing before the ready time and must finish before the deadline. Each thread is an implicitly observable action, and hence may have a ready time and a deadline.

Apart from the ready time and deadline constraints, programs in Maruti can also specify *relative* timing constraints, those which constrain the interval between two events. For each action, the start and end of the action mark the observable events. A relative constraint is used to constrain the temporal separation between two such events. It may be a relative deadline constraint which specifies the upper bound on time between two events, or a delay constraint which specifies the lower bound on time between the occurrence of the two events. The interval constraints are closer to the event-based real-time specifications, which constrain the minimum and/or maximum distance between two events and allow for a rich expression of timing constraints for real-time programs.

386

- Replication and Fault Tolerance. At the application level fault tolerance is achieved by creating resilient applications by replicating parts, or all, of the application. The configuration language eases the task of achieving fault tolerance by allowing mechanisms to replicate the modules, and services, thus achieving the desired amount of resiliency. By specifying allocation constraints, a programmer can ensure that the replicated modules are executed on different partitions.

5. Analysis and Resource Allocation

This phase involves analyzing the resource allocation and scheduling of a collection of applications in terms of their real-time and fault-tolerance properties. The properties of the system are analyzed with respect to the system configuration and the characteristics of the runtime system, and resource calendars are generated.

The analysis phase converts the application program into fine-grained segments called *elemental units*. All subsequent analysis and resource allocation are based on EUs.

5.1. Elemental Unit Model

The basic building block of the Maruti computation model is the elemental unit (EU). In general, an elemental unit is an executable entity which is triggered by incoming data and signals, operates on the input data, and produces some output data and signals. The behavior of an EU is atomic with respect to its environment. Specifically:

- All resources needed by an elemental unit are assumed to be required for the entire length of its execution.

- The interaction of an EU with other entities of the systems occurs either before it starts executing or after it finishes execution.

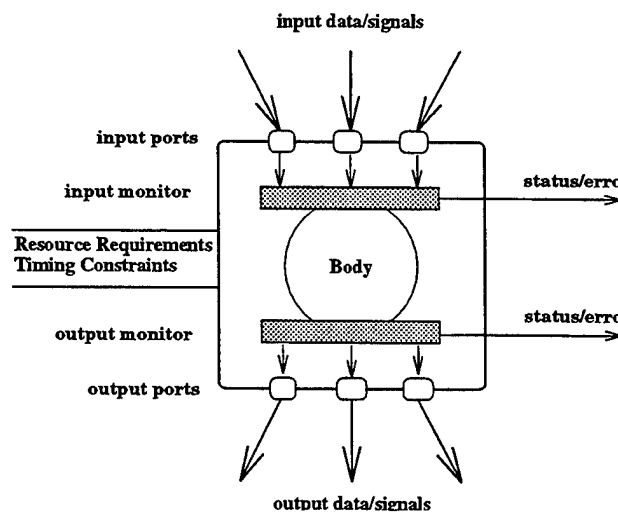The components of an EU are illustrated in Figure 1 and are described below:

Figure 1: Structure of an Elemental Unit

387

- **Input and Output Ports.** Each EU may have several input and/or output ports. Each port specifies a part of the interface of the EU. The input ports are used to accept incoming input data to the EU, while the output ports are used for feeding the output of the EU to other entities in the system.

- **Input and Output Monitors.** An input monitor collects the data from the input ports, and provides it to the main body. In doing so, it acts as a filter, and may also be used for error detection and debugging. The input monitors are also used for supporting different triggering conditions for the EU. Similar to input monitors, the output monitors act as filters to the outgoing data. The output monitor may be used for error detection and timingconstraint enforcement. The monitors may be connected to other EUs in the system and may send (asynchronous) messages to them reporting errors or status messages. The receiving EU may perform some error-handling functions.

- **Main Body.** The main body accepts the input data from the input monitor, acts on it, and supplies the output to the output monitor. It defines the functionality provided by the EU.

Annotated with an elemental unit are its resource requirements and timingconstraints, which are supplied to the resource schedulers. The resource schedulers must ensure that the resources are made available to the EU at the time of execution and that its timing constraints are satisfied.

5.2. Composition of EUs

In order to define complex executions, the EUs may be composed together and properties specified on the composition. Elemental units are composed by connecting an output port of an EU with an input port of another EU. A valid connection requires that the input and output port types are compatible, i.e., they carry the same message type. Such a connection marks a one-way flow of data or control, depending on the nature of the ports. A composition of Eus can be viewed as a directed acyclic graph, called an *elemental unit graph* (EUG), in which the nodes are the EUs, and the edges are the connections between EUs. An incompletely specified EUG in which all input and output ports are not connected is termed as a *partial EUG* (PEUG). A partial EUG may be viewed as a higher level EU. In a complete EUG, all input and output ports are connected and there are no cycles in the graph. The acyclic requirement comes from the required time determinacy of execution. A program with unbounded cycles or recursions may not have a temporally determinate execution time. Bounded cycles in an EUG are converted into an acyclic graph by loop unrolling.

The composition of EUs supports higher level abstractions and the properties associated with them. By carefully choosing the abstractions, the task of developing applications and ensuring that the timing and other operational constraints are satisfied can be greatly simplified. In Maruti, we have chosen the following abstractions:

- *A thread* is a sequential composition of elemental units. It has a sequential flow of control which is triggered by a message to the first EU in the thread. The flow of control is terminated with the last EU in the thread. Two adjacent EUs of a thread are connected by a single link carrying the flow of control. The component elemental units may receive messages or send messages to elemental units outside the thread. All EUs of a thread share the execution stack and processor state.

- *A job* is a collection of threads which cooperate with each other to provide some functionality. The partial EUGs of the component threads are connected together in a well defined manner to form a complete EUG. All threads within a job operate under a global timing constraint specified for the job.

5.3. Program Analysis

Program modules are independently compiled. In addition to the generation of the object code, compilation also results in the creation of partial EUGs for the modules, i.e., for the services and entries in the module, as well as the extraction of resource requirements such as stack sizes for threads, memory requirements, and logical resource requirements.

Invocation of an entry point and service call starts a new thread of execution. A control flow graph is generated for each service and entry. The control flow graph and the MPL primitives are used to delineate EU boundaries. Note that an EU execution is atomic, i.e., all resources required by the EU are assumed to be used for the entire duration of its execution. Further, all input messages are assumed to be logically received at the start of an EU and all output messages are assumed to be logically sent at the end of an EU. At compilation time, the code for each entry and service is broken up into one or more elemental units. The delineation of EU boundaries is done in a manner that ensures that no cycles are formed in the resultant EUG. Thus, for instance, a send followed by a receive within the same EU may result in a cyclic precedence and must be prevented. We follow certain rules of thumb to delineate EU boundaries, which may be overridden and explicitly changed by the user. The EU boundaries are created at a receive statement, the beginning and end of a resource block, and the beginning and end of an observable action. For each elemental unit a symbolic name is generated and is used to identify it. The predecessors and successors of the EU as well as the source code line numbers associated with the EU are identified and stored. The resource and timing requirements that can be identified during compilation are also stored, and place holders are created for the remaining information.

Given an application specification in the Maruti Configuration Language and the component application modules, the integration tools are responsible for creating a complete application program and extracting out the resource and timing information for scheduling and resource allocation. The input to the integration process are the program modules, the partial EUGs corresponding to the modules, the application configuration specification, and the hardware specifications. The outputs of the integration process are: a specification for the loader for creating tasks, populating their address spaces, creating the threads ad channels, and initializing the task; loadable executables of the program; and the complete application EUG along with the resource descriptions for the resource allocation and scheduling subsystem.

5.4. Communication Model

Maruti supports message passing and shared memory models for communication.

- **Message Passing.** Maruti supports the notion of one-way message passing between elemental units. Message passing provides a location-independent and architecture-transparent communication paradigm. A channel abstraction is used to specify a one way message communication path between a sender and a receiver. A one-way message-passing channel is set up by declaring the output port on the sender EU, the input port on the receiver EU, and the type of message. The communication is asynchronous with respect to the sender, i.e., the sender does not block.

- **Synchronous Communication.** Synchronous communication is used for tightly coupled message passing between elemental units of the same job. For every invocation of the sender there is an invocation of the receiver which accepts the message sent by the sender. The receiver is blocked (de-scheduled) until message arrival under normal circumstances. The messages in a synchronous communication channel are delivered in FIFO order.

- **Asynchronous Communication.** Asynchronous communication may be used for message passing between elemental units not belonging to the same job. It may also be used between real-time and non-real-time jobs. In such communication, neither the sender nor the receiver is blocked (i.e., there is no synchronization). Since the sender and receiver may execute at different rates, it is possible that no finite amount of buffers suffice. Hence, an asynchronous communication channel is inherently lossy. The receiver may specify its input port to be inFirst or inLast to indicate which messages to drop when the buffers are full. The first message is dropped in an inLast channel, while the last message is dropped in an inFirst channel.

There may be multiple receivers of a message, thus allowing for multi-cast messages. Similar to a one-to-one channel, a multicast channel may also be synchronous or asynchronous. All receivers of a multi-cast message must be of the same type.

- **Shared Memory.** Shared memory is also supported in Maruti. The simplest way to share memory between EUs is to allow them to exist within the same address space. We use task abstraction for this purpose. A task consists of multiple threads operating within it, sharing the address space. The task serves as an execution environment for the component threads. A thread may belong to only one task. In addition to the shared memory within a task, inter-task sharing is also supported through the creation of shared memory partitions. A shared memory partition is a shared buffer which can be accessed by any EU permitted to do so. The shared memory partitions provide an efficient way to access data shared between multiple EUs. The shared memory communication paradigm provides just the shared memory - it is the user's responsibility to ensure safe access to the shared data. This can be done by defining a logical resource and ensuring that the resource is acquired every time the shared data is accessed. By providing appropriate restrictions on the logical resource, safe access to data can be ensured.

5.5. Resource Model

A distributed system consists of a collection of autonomous processing nodes connected via a local area network. Each processing node has resources classified as processors, logical resources, and peripheral devices. Logical resources are used to provide safe access to shared data structures and are passive in nature. The peripheral devices include sensors and actuators. Restrictions may be placed on the preemptability of resources to maintain resource consistency. The type of the resource determines the restrictions that are placed on the preemptability of the resource and serves to identify operational constraints for the purpose of resource allocation and scheduling. We classify the resources into the following types based on the restrictions that are imposed on their usage.

- **Non-preemptable.** The inherent characteristics of a resource may be such that it prevents preemptability, i.e., any usage of the resource must not be preempted. Many devices require non-preemptive scheduling. For resources which require the use of CPU, this implies non-preemptive execution from the time the resource is acquired until the time the resource is released.

- **Exclusive.** Unlike a non-preemptive resource, an exclusive resource can be preempted. However, the resource may not be granted to anyone else in the meantime. A critical section is an example of a resource which must be used in exclusive mode.

- **Serially Reusable.** A serially reusable resource can not only be preempted but may also be granted to another EU. The state of such resources can be preserved and restored when the resource is granted back.

- **Shared.** A shared resource may be used by multiple entities simultaneously. In a single processor system, since only one entity is executing at a given time, there is no distinction between a shared resource and a serially reusable resource.

A non-preemptable resource is the most restrictive and a shared resource is the least restrictive in terms of the type of usage allowed. An application requesting the use of a resource must specify when the resource is to be acquired, when it is to be released, and the restrictions on the preemptability of the resource. The resource requirements for applications may be specified at different levels of computational abstractions as identified below.

- **EU level.** The lowest level a resource requirement can be specified at is the EU level. A resource requirement specified at the EU level implies that the resource is acquired and released within the EU. For scheduling purposes, it is assumed that the resource is required for the entire duration of the execution of the EU.

- **Thread Level.** Resource specification at the thread level is used for resources which are acquired and released by different EUs belonging to the same thread. For instance, a critical section may be acquired in one EU and released in another one.

- **Job Level.** Job-level resource specifications are used to specify resources which are not acquired and released for each invocation of a periodic or sporadic job. Instead, these resources are acquired at the job initialization and released at job termination. For a periodic job, an implicit resource associated with each thread are the thread data structures (including procesor stack and registers).

5.6. Operational Constraints

The execution of EUs is constrained through various kinds of operational constraints. Such constraints may arise out of restricted resource usage or through the operational requirements of the application. Examples of such constraints are: precedence, mutual exclusion, ready time, and deadline. They may be classified into the following categories:

- **Synchronization Constraints.** Synchronization constraints arise out of data and control dependencies or through resource preemption restrictions. Typical examples of such constraints are precedence and mutual exclusion.

- **Timing Constraints.** Many types of timing constraints may be specified at different levels, i.e., at job level, thread level, or EU level. At the job level, one may specify the ready time, deadline, and whether the job is periodic, sporadic, or aperiodic. For threads, a ready time and deadline may be specified relative to the job arrival. Likewise, a ready time and deadline may be specified for an individual EU. We also support the notion of relative timing constraints, i.e., constraints on the temporal distance between the execution of two EUs.

- **Allocation Constraints.** In our model, tasks are allocated to processing nodes. Allocation constraints are used to restrict the task allocation decisions. Allocation constraints often arise due to fault-tolerance requirements, where the replicas of EUs must be allocated on different processing nodes. Similarly, when two tasks share memory, they must be allocated on the same processing node. Sometimes a task must be bound to a processing node since it uses a particular resource bound to the node (e.g., a sensor).

The operational constraints are made available to the resource allocation and scheduling tools which must ensure that the allocation and scheduling maintains the restrictions imposed by the constraints. The model does not place any a priori restrictions on the nature of the constraints that may be specified. However, the techniques used by the resource allocator and scheduler will depend on the type of constraints that can be specified.

5.7. Allocation and Scheduling

After the application program has been analyzed and its resource requirements and execution constraints identified, it can be allocated and scheduled for a runtime system.

This final phase of program development depends upon the physical characteristics of the hardware on which the application will be run, for example, the location of devices and the number of nodes and type of processors on each node in the distributed system.

Maruti uses time-based scheduling and the scheduler creates a data structure called a *calendar* which defines the execution instances in time for all executable components of the applications to be run concurrently.

We consider the static allocation and scheduling in which a task is the finest granularity object of allocation and an EU instance is the unit of scheduling. In order to make the execution of instances satisfy the specifications and meet the timing constraints, we consider a scheduling frame whose length is the least common multiple of all tasks' periods. As long as one instance of each EU is scheduled in each period within the scheduling frame and these executions meet the timing constraints a feasible schedule is obtained.

As a part of the Maruti development effort, a number of scheduling techniques have been developed and are used for generating schedules and calendars for task sets. These techniques include the use of temporal analysis and simulated annealing. Schedules for single-processor systems as well as multiple-processor networks are developed using these techniques.

6. Maruti Runtime System

The runtime system provides the conventional functionality of an operating system in a manner that supports the timely dispatching of jobs. There are two major components of the runtime system - the *Maruti core*, which is the operating system code that implements scheduling, message passing, process control, thread control, and low level hardware control, and the *runtime dispatcher*, which performs resource allocation and scheduling for dynamic arrivals.

6.1. The Dispatcher

The dispatcher carries out the following tasks:

- **Resource Management.** The dispatcher handles requests to load applications. This involves creating all the tasks and threads of the application, reserving memory, and loading the code and data into memory. All the resources are reserved before an application is considered successfully loaded and ready to run.

- **Calendar Management.** The dispatcher creates and loads the calendars used by applications and activates them when the application run time arrives. The application itself can activate and deactivate calendars for scenario changes.

- **Connection Management.** A Maruti application may consist of many different tasks using channels for communication. The dispatcher sets up the connections between the application tasks using direct shared buffers for local connections or a shared buffer with a communications agent for remote connections.

- **Exception Handling.** Rogue application threads may generate exceptions such as missed deadlines, arithmetic exceptions, stack overflows, and stray accesses to unreserved memory. These exceptions are normally handled by the dispatcher for all the Maruti application threads. Various exception handling behaviors can be configured, from terminating the entire application or just the errant thread, to simply invoking a task-specific handler.

6.2. Core Organization

The core of the Maruti hard real-time runtime system consists of three data structures:

- *The calendars* are created and loaded by the dispatcher. Kernel memory is reserved for each calendar at the time it is created. Several system calls serve to create, delete, modify, activate, and deactivate calendars.

- *The results table* holds timing and status results for the execution of each elemental unit. The maruti_calendar_results system call reports these results back up to the user level, usually to the dispatcher. The dispatcher can then keep statistics or write a trace file.

- *The pending activation table* holds all outstanding calendar activation and deactivation requests. Since the requests can come before the switch time, the kernel must track the requests and execute them at the correct time in the correct order.

393

The scheduler gains control of the CPU at every clock tick interrupt. At that time, if a Maruti thread is currently running and its deadline has passed its execution is stopped and an exception raised.

If any pending activations are due to be executed those requests are handled, thereby changing the set of active calendars. Then the next calendar entry is hecked to see if it is scheduled to execute at this time. If so, the scheduler switches immediately to the specified thread. If no hard real-time threads are scheduled to execute, the calendar scheduler falls through to the soft and non-real-time, priority-based schedulers.

Maruti threads indicate to the scheduler that they have successfully reached the end of their elemental unit with the maruti_unit_done system call. This call marks the current calendar entry as done and fills in the time actually used by the thread. The Maruti thread is then suspended until it next appears in the calendars. Soft and non-real-time threads can be run until the next calendar entry is scheduled and are executed using a priority based scheduling for the available time slots.

At all times the Maruti scheduler knows which calendar entry will be the next one to run so that the calendars are not continually searched for work. This is recalculated when maruti_unit_done is called or whenever the set of active calendars changes.

6.3. Multiple Scenarios

The Maruti design includes the concept of scenarios, implemented at runtime as sets of alternative calendars that can be switched quickly to handle an emergency or a change in operating mode. These calendars are pre-scheduled and able to begin execution without having to invoke any user-level machinery. The dispatcher loads the initial scenarios specified by the application and activates one of them to begin normal execution. However, the application itself can activate and deactivate scenarios. For example, an application might need to respond instantaneously to the pressing of an emergency shutdown button. A single system call then causes the immediate suspension of normal activity and the running of the shutdown code sequence. Calendar activation and deactivation commands can be issued before the desired switch time. The requests are recorded and the switches occur at the precise moment specified. This allows the application to insure smooth transitions at safe points in the execution.
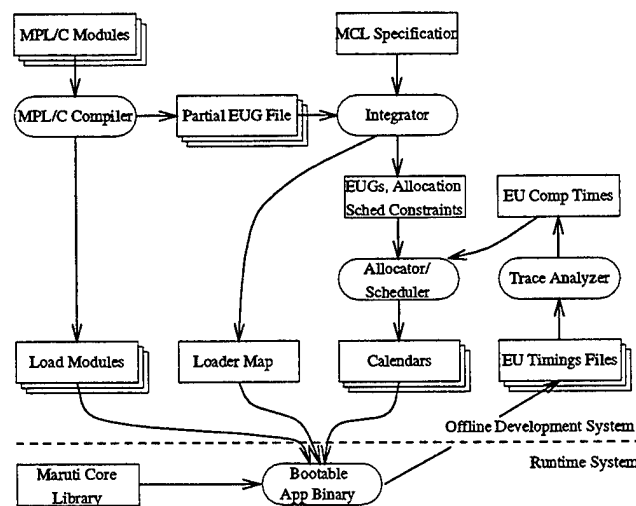

Figure 2: Maruti System Architecture

394

7. Maruti 3.1 System Architecture

Maruti 3.1, the current version of the operating system, implements most of the above design with a series of development tools that operate in a Berkeley Unix development environment (NetBSD 1.0) on IBM-compatible 486 or Pentium PCs. Maruti applications can be run stand-alone on the bare hardware or under a Unix-based debugging environment.

- MPL code is processed by the MPL compiler, a modified version of the GNU C compiler. The MPL compiler generates both the compiled object code and partial EUG file that contains all information extracted from the module for further analysis, including the boundaries of the elemental units of the program.

- The application's MCL specification is read and interpreted by the *integrator*. The PEUG file describing each module used in the application is processed and intermodule type checking performed. The integrator generates a file specifying the full application EUG, allocation, and scheduling constraints.

- The allocator/scheduler reads in the data supplied by the integrator and a description of the physical system on which to allocate the application. The allocator searches for an arrangement of elemental units on the nodes of the network that satisfies all the timing and allocation constraints, considering the computation times for each elemental unit. If a feasible schedule can be found, a *calendar file* for each resource is generated. A loader map is also generated which describes, for the runtime system, each task, thread, shared memory area, and communications link so that all the resources can be reserved when the application is loaded.

- The computation time analyzer takes timing trace information generated by the runtime system and generates worst-case execution times for all the Eus of the application. This timing information can be used in subsequent runs of the scheduler to refine the schedule and verify its feasibility given changes in computation times. Use of the timing tool during testing leads to very high confidence in the schedule.

7.1. Runtime Environments

Compiled and analyzed Maruti applications can be executed in multiple runtime environments.

- The *Maruti/Virtual* runtime environment allows the debugging of Maruti applications within the development environment. Applications run in virtual real-time under Unix, allowing temporal debugging, including single stepping the real-time calendars.

- The *Maruti/Mach* runtime environment is a modified version of Mach which allows the running of real-time Maruti programs within the Mach environment, where the real-time and non-real-time task can co-exist and interact in the same host.

- The *Maruti/Standalone* runtime environment runs the application on the bare hardware, suitable for embedded systems. The application is linked with a minimal Maruti core library and can be booted directly.

395

7.2. Maruti/Virtual Runtime Environment

Testing real-time programs in their native embedded environments can be tedious and very time-consuming because of the lack of debugging facilities and the requirement to reload and reboot the target computer every time a change is made. Maruti provides a Unix-based runtime system that allows the execution of Maruti hard-real-time applications from within the Unix development environment. This Unix execution environment supports the following features:

- The Maruti application has direct control of its I/O device hardware.

- Graphical output and keyboard input can go either to the PC console, as in the Maruti/Standalone and Maruti/Mach environments, or appear in an X window on any Unix workstation, possibly across the network.

- The application can be run under the Unix GNU Debugger, allowing the examination of program variables and stack traces, setting of breakpoints, and post-mortem analysis.
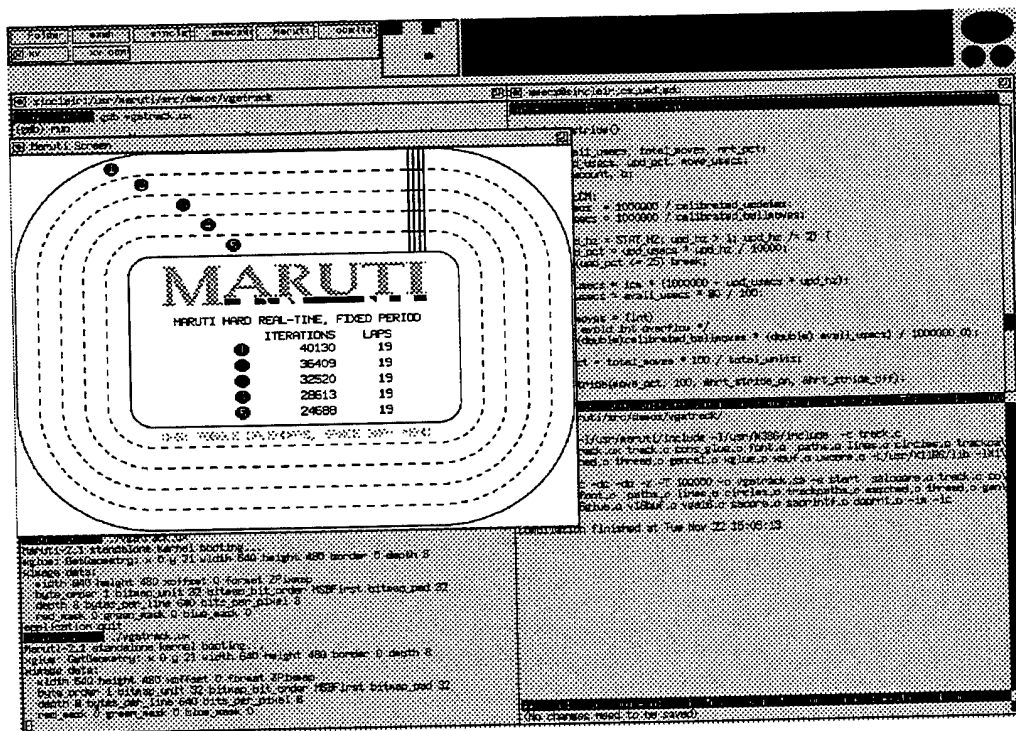

Figure 3: Maruti/Virtual screen running in the development environment

- The application has access to Unix standard output so it can print debug and status messages to the interactive session while running.

- The Maruti application runs in *virtual real-time*; that is, it sees itself running in hard-real-time against a virtual time base.

- The virtual time can be manipulated through the runtime system for temporal debugging. Virtual time can be slowed down or sped up, and individual elemental units (EU) or whole calendars can be single-stepped or traced.

7.3. Maruti/Standalone Real-Time Environment

- Maruti/Standalone provides a minimal runtime system for the execution of a Maruti application on the bare hardware. The stand-alone environment has the following attributes:

- The stand-alone version of an application is built from the same object modules as are used in the Unix and Maruti/Mach execution environments.

- All the modules of the application are bound with only those routines of the Maruti core that are needed into one executable, suitable for booting directly or converting into ROM.

- The application has complete control of the computer hardware.

- The application runs in hard real-time with very low overhead and variability.

- The minimal Maruti/Standalone core library currently consists of about 16 KB of code and 16 KB of data.

- The optional Maruti Distributed Operation support (including network driver) is about 14 KB of code and 9 KB of data.

- The optional Maruti graphics package currently consists of, for the standard VGA version, 10 KB of code and 20 KB of data (plus 150K for a secondary frame buffer for best performance).

7.4. Maruti/Mach Real-Time Environment

The original execution environment for Maruti-2 was a modified version of the CMU Mach 3.0 kernel. Maruti/Mach is potentially useful in hybrid environments in which the real-time components co-exist with Mach and Unix processes on the same CPU. Because of preemptability problems in CMU Mach we will not be distributing Maruti/Mach until it can be rehosted onto OSF1/MK real-time kernels.

The Maruti/Mach features include the following:

- A calendar-based real-time scheduler has been added to the CMU Mach 3.0 kernel. This scheduler takes precedence over the existing Mach scheduler, running Maruti elemental units from the calendar at the proper release time.

- The Maruti application and most of the runtime system run as normal Mach user-level tasks and threads, which are wired down in memory.

397

- The Maruti application may communicate with non-Maruti Unix and Mach processes through shared memory.

- The Maruti/Mach kernel maintains runtime information for each elemental unit executed, and makes that information available to the user-level code for worst-case computation time analysis.

- Parts of the CMU Mach kernel remain unpreemptable. Nevertheless, on a dedicated system we can achieve release time variability of about 100 microseconds. The context switch time is about 200 microseconds.

- The new release of OSF Research Institute Mach MK6.0 addresses most of the Mach kernel preemptability concerns. We will be porting Maruti/Mach to this base in the near future.

8. Future Directions

The Maruti Project is an ongoing research effort. We hope to extend the current system in a number of possible directions. Of course, since this is a research project, we expect our ideas to evolve over time as we gain experience and get feedback from users.

8.1. Scheduling and Analysis Extensions

Preemptable Scheduling of Hard-Real-Time Tasks

We are planning to extend our scheduling approach to incorporate controlled preemptions of tasks. To date we have concentrated on using non-preemptable executions of tasks, which simplifies scheduling and eases exclusion problems in application development. However, the non-preemptability assumption to exclusion is not scalable to a multiprocessor, as threads running on different processors can interfere with each other. Controlled preemption is more powerful, as it allows scheduling of long-running tasks concurrently with high frequency tasks. Preemption will remain under the control of the application.

Language support for atomic actions will be developed to replace the assumption of non-preemptable EU's. Action statements will serve to delineate sections of code on which precise timing requirements can be imposed by the application designer. Combined with critical region statements (already implemented), actions will allow the programmer to specify precisely the desired timing and resource interrelationships in a manner that is scalable to a multiprocessor or network cluster, unlike the non-preemptability assumption.

We will extend the Maruti run-time system to handle preemptable hard real-time tasks. This will be done in coordination with the analysis tools which will generate multiple calendar entries for the preempted EUs. All but the last entry for the EU will be marked as preemptable, and all but the first will be marked as continuation entries. This is enough information for the run-time scheduler to correctly handle the preemption in a controlled manner, even when the EU completes early.

Integration of Time-based and Priority-Based Scheduling

We plan to integrate the time-based and priority-based scheduling in a single framework. To date we have concentrated on time-based scheduling only. To support other scheduling paradigms within the time-based framework, we may reserve time slots in the schedule and associate a queue of waiting tasks which are executed on the basis of their priorities. In this way we can implement rate-monotonic style static priority schemes as well as Earliest-Deadline-First style dynamic priority schemes within the Maruti framework. However, in order to assure that the tasks executed under priority-based scheduling will continue to meet their temporal requirements, extensions to the analysis techniques are required. We will develop analysis techniques suitable for this purpose.

We will extend the Maruti implementation to support non-calendar schedulers, such as priority based or earliest-deadline-first based schedulers. These schedulers will run in particular slots specified in the Maruti calendar, or when the calendar is idle.

POSIX-RT Subset API

In a related area, we plan to study the use of a subset of the POSIX API as the Maruti API for soft and non-real-time tasks. We will implement as much of the POSIX-RT API as is appropriate and practicable.

Asynchronous Events

Generally, in a time-based system, events are polled for at the maximum frequency at which they are expected. This type of event handling is easy to analyze within the time-based framework, and makes explicit the need to reserve enough time to handle the event stream at its worst-case arrival rate. At this worst-case rate, polling is more efficient than interrupt-driven event handling because the interrupt overhead is avoided. However, at low event rates, polling is less efficient and fragments the cpu idle time (where we define idle time from the point of view of hard real-time tasks). While conservation of idle time is not an issue for small controllers, it becomes very important when there are soft- and non-real-time tasks running in the system.

Currently, Maruti takes the polling approach to ease analysis and to better handle the worst case rate. We plan to study the analysis required to accommodate asynchronous events within a calendar schedule. Our intended approach is to work with a specified maximum frequency, relative deadline, and computation time of the asynchronous event, and to reserve enough time in the calendar for the event to occur at its maximum frequency.

We will extend the Maruti run-time system to register and dispatch event handlers in response to external events. Included in this extension will be the ability to detect and appropriately handle overload conditions (i.e. when the events occur more quickly that expected).

Multi-Dimensional Resource Scheduling Research

A typical real-time application requires several resources for it to execute. While CPU is the most critical resource, others have to be made available in a timely manner. Generation of schedules for multiple resources is known to be a difficult problem. Our approach to date has been to develop efficient search techniques, such as one based on simulated annealing.

Realistic problems contain a variety of interdependencies among tasks which must be reflected as constraints in scheduling. We plan to develop efficient techniques for scheduling the allocation and deallocation of portions of multidimensional resources. In particular, we will address the problems of allocation and management of resources such as memory and disk space, that can accommodate many entities simultaneously.

Scheduling System-Specific Topologies

In a related area, many communications networks have more complex structures than a simple bus and cannot be treated as a single dedicated resource. We will study the extension of our scheduling algorithms to support point-to-point meshes of nodes (with store-and-forward of messages), switched networks (such as MyriNet), and sophisticated backplanes such as that used in the Intel Paragon.

We will investigate the use of a general framework for specifying the properties of connection topologies to the Maruti scheduler. In the worst cases, the scheduler for a complex interconnection technology may have to be programmed explicitly. To handle such cases, we will develop a modular interface into our allocator/scheduler into which such backplane-specific schedulers can be plugged.

Static Estimation of Execution Times

Currently, execution times are derived through extensive testing of the program on the target hardware environment. Deriving the execution time through static analysis is hampered by the data dependencies present in large number in most programs.

We will investigate the use of static analysis to help prove the execution time limits of programs. While generating a reasonable computation time estimate through static analysis is not feasible in general, it is possible to get accurate results for large segments of a program, and to clearly identify the existing data dependencies so that the programmer can-through program modifications or directives to the analysis tool-eliminate, curtail, or characterize the data dependencies well enough to get very useful verification of the time properties of the program.

Temporal Debugging

When we develop real-time applications we need techniques for observing the temporal behavior of programs. For their functional characteristics we can use standard debuggers which permit the observation of the state of execution at any stage. This, however, destroys the temporal relationships completely. In Maruti/Virtual we provide the facilities of controlling the execution of all parts of an application with respect to a virtual time which advances under the control of keyboard directives. Thus we can pause the execution at any virtual time instant with the assurance that all temporal relationships with respect to this instant are accurately reflected in the state of the program. We use the term temporal debugging for this.

We will conduct research on the theoretical aspects of the issues of temporal debugging and consider the implications of temporal debugging. In particular, we will study how the interactions of programs executing in virtual time with external events which occur with respect to their own time line should be captured in temporal debugging. We will also study how the virtual times of several nodes in a distributed environment should be coordinated.

We will extend our implementation of temporal debugging tools in the Maruti/Virtual environment to support temporal debugging of distributed programs, and to support fine grained modification of the time line.

Dynamic Schedule Generation

We will develop the notion of time horizons to support controlled modifications of the hard real-time calendars at runtime to support programs that generate schedules dynamically. While the run-time mechanisms for modifying the calendars are already implemented, research issues relating to finding safe points to switch schedules, and scheduling the schedulers themselves, have to be studied before effective use can be made of on-line calendar generation.

8.2. Fault Tolerance

Maruti currently supports several powerful mechanisms for building fault tolerant applications:

- Maruti Configuration Language (MCL) constructs allow the application designer to specify replication of application subsystems with forkers and joiners inserted into the communication streams, as well as the allocation constraints necessary to correctly partition the replicated subsystems for the desired level of fault tolerance.

- Maruti Programming Language (MPL) allows the programming of application specific fault tolerance components such as forkers and joiners, elemental unit monitors, and channel monitors.

- The run-time system supports multiple calendars, allowing the application to switch to emergency or fault handling scenarios in real time.

   We plan to extend the existing mechanisms by providing tools and new mechanisms to better automate the process of building fault tolerant applications. The new features will include:

- A library of forkers and joiners that can be incorporated into applications.

- Support for multicast messages.

- Better support in Maruti Programming Language (MPL) for EU and channel monitors.

- Automatic replication of subsystems, and analysis of fault tolerance properties through MAGIC, the graphical integrator described below.

8.3. Clock Synchronization

Currently, distributed Maruti handles clock drift at boot-up time, and thereafter time slave nodes simply adopt the time-master's clock periodically. This scheme is suitable for many applications, but is not ideal for embedded control systems that will suffer from a discontinuous time jump.

   To address this problem we plan to develop and implement time-synchronization algorithms that operate concurrently with the distributed real-time program to continually adjust the clocks on all the nodes, taking into account changes in their relative drift. This

will most likely involve a regular time pulse from a master clock, from which the other nodes continually measure their drift and fine-tune their tick rates. Since the clock drifts are about one order of magnitude less than the communication latency variances, a simple algorithm will not suffice here.

8.4. Heterogeneous Operation

We will extend our communications agents and boot protocol to translate typed Maruti messages between heterogeneous hosts when needed. The off-line Maruti analysis tools already collect information on the types of the channel endpoints for type-checking the connection. We will carry this information through to the run-time system for use in those channels that are connected between heterogeneous nodes.

8.5. MPL/Ada

We will incorporate Maruti Programming Language (MPL) features and analysis into the Ada 95 programming language as we did for ANSI C in the current MPL, which we will now refer to as MPL/C. Implementing MPL/Ada will involve the following tasks:

- A detailed design review studying those features of Ada which are compatible with Maruti and those that are not, and how best to proceed with the implementation of MPL/Ada.

- Port GNU Ada (GNAT) to our NetBSD development environment.

- Implement as much of the Ada run-time environment as is practicable on the Maruti run-time.

- Install hooks into GNAT to extract the resource usage information we need. We expect this work will leverage heavily from the MPL/C work, as GNAT is derived from the same back-end code base as GNU C.

- Develop and enforce within GNAT those restrictions on Ada constructs needed in order to preserve the properties needed for our hard real-time analysis.

- Add support for Maruti primitives to the language. Some Maruti primitives might be implementable directly through existing Ada facilities and thus will not require language extensions.
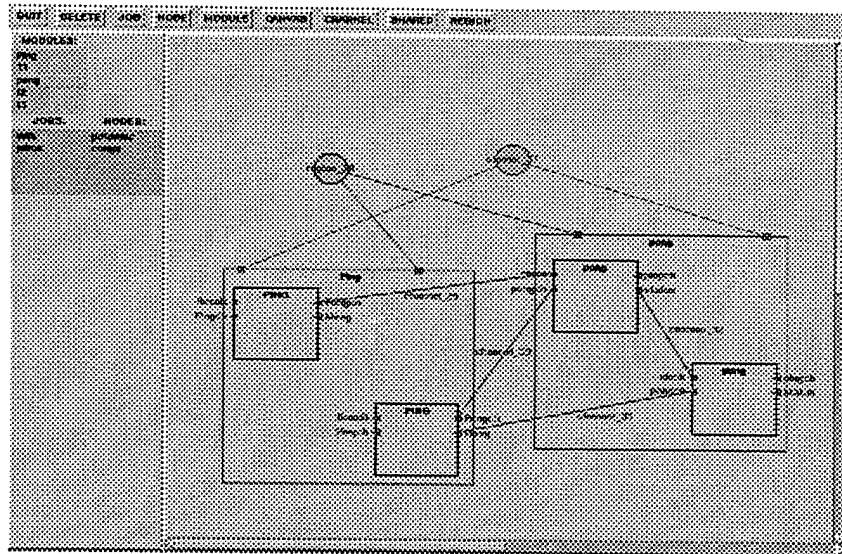
Figure 4: Prototype Graphical Program Integrator Tool.

8.6. Graphical Tools

Graphical Program Development Tools

Currently, Maruti applications are pulled together by an MCL specification, which takes the form of a procedural language whose primitive operations instantiate and bind together the parts of the application. This type of specification language is complete, allowing the specification of large, complex applications connected in arbitrary ways. However, such completeness makes MCL relatively low-level and tedious to program.

We are developing graphical program development tools which allow the application designer to pull together the modules using an entirely graphical user interface-avoiding MCL programming. The on-screen representation of modules can be interconnected with channels and grouped into hierarchical subsystems. The application designer will be able to zoom in and out to view the application at several levels.

The graphical environment will allow both the integration of existing modules and the development of the interfaces of modules that have not yet been written. The tools will generate template MPL code for those modules. In this way the graphical environment functions as a design tool and program generator as well as an integration environment.

The graphical environment will have fault-tolerance analysis built into it. Single points of failure will be identified on-screen. The user will be able to replicate entire subsystems at once, with the *forker* and *joiner* modules and allocation constraints introduced into the application automatically by the system.

This graphical style of application integration will greatly facilitate the building and deployment of reusable software components modules built to be easily customized and reintegrated into many applications. Given a suitable library of reusable component modules and the graphical integrator, it will be possible for non-programmers to build large custom applications from these parts.
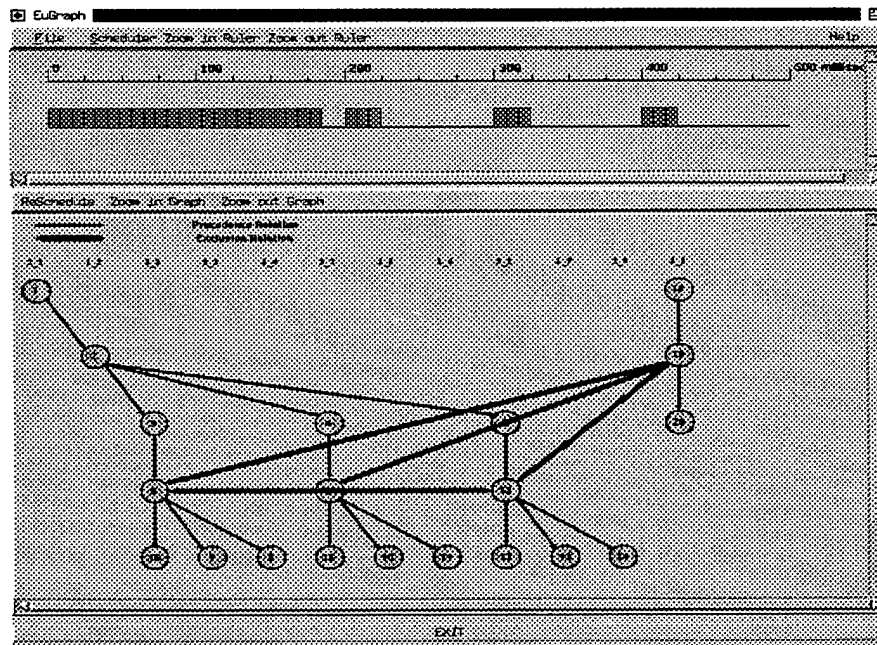
403

Figure 5: Prototype Graphical Resource Scheduling Tool

Graphical Resource Management Tools

Along with the graphical software development tools, we are pursuing graphical resource management tools. These are a non-programmer's interface into the advanced Maruti scheduling technology. The Maruti allocator/scheduler works with the abstract concepts of schedulable entities, available resources, and various types of constraints on the placement of entities and resources. In the Maruti operating system, the scheduling entities are EUs, and the resources are CPUs, network, memory, and devices - but in fact any type of entity or resource can be manipulated by the allocator/scheduler.

A graphical resource management tool will allow the specification of these entities, resources, and constraints on screen in a way more oriented towards the general user. With this tool users should be able to use Maruti scheduling technology to schedule classes, busses, or projects, for example.

We have built a small prototype of the graphical resource manager. The prototype displays the EU graph input to the scheduler as well as the calendar output of the scheduler. The user can edit the EU graph and its constraints and reschedule with the click of a button. The resulting resource calendar is redisplayed.

9. Availability

We are pleased to announce the availability of the Maruti 3.1 Hard Real-Time Operating System and Development Environment.

With Maruti 3.1, we are entering a new phase of our project. We have an operating system suitable for field use by a wider range of users, and we are embarking on the integration of our time-based, hard real-time technology with industry standards and more traditional event-based soft- and non-real-time systems. For this, we are greatly interested in the feedback from users as to the direction of evolution of the system.

404

For the Maruti 3.1 project, we will be pursuing the integration of a POSIX interface for soft and non-real-time applications, the use of Ada for Maruti programming, support for asynchronous events and soft/non-real time schedulers within the time-based framework, and heterogeneous Maruti networks.

For this user-oriented phase of the project we will be making regular releases of our software available to allow interested parties to track and influence our development. To begin this phase we are making our current base hard real-time operating system and its development environment available. This is an initial test release.

Maruti 3.1 will be made available to interested parties on request, via Internet ftp. Please send electronic mail to maruti-dist@cs.umd.edu for details. More information about the Maruti Project, as well as papers and documentation, are available via the World Wide Web at:

        http://www.cs.umd.edu/projects/maruti/|

9.1. Runtime System

The Maruti 3.1 embeddable hard real-time runtime system for distributed and single-node systems includes the following features:

- The core Maruti runtime system is small - 16 KB code for the single node core, 30 KB code for the distributed core.

- The core provides a calendar-based scheduler, threads, distributed message passing using Time Division Multiplexed Access (TDMA) over the network, and tight time synchronization between network nodes.

- Also included in the runtime system is a graphics library suitable for system monitoring displays as well as simulations.

- Maruti runs on PC-AT compatible computers using the Intel i386 (with i387 coprocessor), i486DX, or Pentium processors. Distributed operation currently requires a 3Com 3c507 ethernet card. The graphics library supports standard VGA and Tseng-Labs ET-4000-based Super-VGA. Support for other SVGA chipsets is forthcoming soon.

9.2. Development Environment

Maruti 3.1 includes a complete development environment for distributed embedded hard real-time applications. The development environment runs on NetBSD Unix and includes the following:

- d The Maruti/Virtual debugging environment - simulates the Maruti runtime system within the development environment. The system clock in this environment tracks virtual time, which can be sped up, slowed down in relation to the actual time, or single-stepped or stopped. This allows temporal debugging of the application. Within Maruti/Virtual traces of the application scheduling and network traffic can be monitored in the debugging session.

- The ANSI-C based Maruti Programming Language (MPL/C). MPL adds modules, message passing primitives, shared memory, periodic functions, message-invoked functions, and exclusion regions to ANSI C. MPL is processed by a version of the GNU C compiler which has been modified to recognize the new MPL features, and to output information about the resources used by the MPL program.

- The Maruti Configuration Language (MCL). MCL allows the system designer to specify the placement, timing constraints, and interconnections of all the modules in an application. MCL is a powerful interpreted C-like language, allowing complex, hierarchical configuration specifications, including replication of components and installation-site specific sizing of the application. The MCL processor analyses the application graph for completeness, and type-checks all connections.

- The Maruti Allocator/Scheduler. The Maruti allocation and schedulingtool analyses the information generated by the MPL compiler and the MCL integrator to find an allocation and scheduling of the tasks of a distributed application across the nodes of a Maruti network. All relative and global timing, exclusion, and precedence constraints are taken into account in finding a schedule, as are the network speed and scheduling parameters.

- The Maruti Timing Trace Analyzer. The Timing Analyzer calculates worst-case computation times from timing files output by the runtime system. Computation times are calculated for each scheduling unit in the application, and these times can be fed back into the Allocator/Scheduler for more precise scheduling analysis.

- The Maruti Runtime Binder (mbind). One of the features of Maruti is the late binding of an application to a particular runtime system. The same application binaries can be combined with different system libraries to build a binary customized for a particular application in a particular setting. Only those portions of the system library needed for that binding are included. Mbind manages this final step.

- The Maruti Application Builder (mbuild). Mbuild automates the process of building an application by generating for the programmer a customizable makefile that manages the complete process of compiling, configuring, scheduling, and binding an application.